

RESTful Web APIs

可因时而变的服务

RESTful Web APIs

中文版



[美] *Leonard Richardson & Mike Amundsen* 著

Sam Ruby 序

赵震一 李哲 译

O'REILLY®



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

RESTful Web APIs 中文版

近年来, REST的流行导致了各种“RESTful” API的巨大增长, 但是这些API却错失了很多架构的好处。通过这本实用指南, 你将可以学习到如何设计可用的, 并能随着时间不断进化的REST API。通过专注于跨多种领域的解决方案, 本书向你展示了该如何使用那些为世界上最成功的分布式计算系统——万维网而设计的工具, 从而来创建强大且安全的应用。你将探索REST背后的概念, 学习多种可用于创建基于超媒体API的策略, 并在本书一步步的指导下整合你所学到的所有内容, 从而去设计RESTful的web API。

- 审查了包括集合模式和纯超媒体在内的API设计策略。
- 理解如何将超媒体与表述整合进一个一致的API。
- 探索XMDP和ALPS⁺ profile格式是如何帮助你应对web API的“语义挑战”的。
- 学习近二十种标准化的超媒体数据格式。
- 应用在API实现中使用HTTP的最佳实践。
- 使用JSON-LD标准及其他Linked Data方法来创建web API。
- 理解在嵌入式系统使用REST的CoAP协议。

“这是一本了不起的书！
*RESTful Web APIs*覆盖了当今API领域最重要的趋势和实践。”

—— John Musser
ProgrammableWeb创始人

Leonard Richardson, *Ruby Cookbook* (O'Reilly) 一书的作者, 曾创建了包括Beautiful Soup在内的多个开源代码库。

Mike Amundsen是包括《使用HTML5和Node构建超媒体API》(O'Reilly) 在内的十几本为人所称道的技术图书的作者。

Sam Ruby是W3C HTML工作组的联合主席, 同时也是IBM新兴技术组的一名高级技术人员。

图书分类: Web开发

责任编辑: 张春雨

策划编辑: 张春雨



Broadview[®]
WWW.BROADVIEW.COM.CN

www.phei.com.cn

O'REILLY[®]
oreilly.com.cn

ISBN 978-7-121-23115-5



9 787121 231155 >

O'Reilly Media, Inc. 授权电子工业出版社出版

此简体中文版仅限于中国大陆 (不包含中国香港、澳门特别行政区和中国台湾地区) 销售发行

This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

定价: 79.00元

RESTful Web APIs中文版

RESTful Web APIs

[美] Leonard Richardson & Mike Amundsen 著

赵震一 李哲 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书是针对RESTful API的实用指南,通过展示各种用来创建高可用应用的强大工具,讲解REST的深层原理,以及介绍基于超媒体API的策略,使读者得以在将上述内容融会贯通后,设计出让客户高度满意的RESTful的web API。本书极具权威性与前瞻性,既代表了API领域的最前沿趋势,也覆盖了API领域的最重要实践。

本书适合所有从事Web开发和架构工作的读者阅读参考。

©2013 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2014. Authorized translation of the English edition, 2013 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由O'Reilly Media, Inc.授予电子工业出版社。未经许可,不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字:01-2014-2675

图书在版编目(CIP)数据

RESTful Web APIs 中文版 / (美)理查德森(Richardson,L.), (美)阿蒙森(Amundsen,M.)著;赵震一,李哲译. —北京:电子工业出版社,2014.6

书名原文:RESTful Web APIs

ISBN 978-7-121-23115-5

I. ①R… II. ①理… ②阿… ③赵… ④李… III. ①互连网络—网络服务器—程序设计 IV. ①TP368.5

中国版本图书馆CIP数据核字(2014)第087092号

策划编辑:张春雨

责任编辑:张春雨

印 刷:北京丰源印刷厂

装 订:三河市鹏成印业有限公司

出版发行:电子工业出版社

北京市海淀区万寿路173信箱 邮编:100036

开 本:787×980 1/16 印张:26 字数:540.8千字

版 次:2014年6月第1版

印 次:2014年6月第1次印刷

定 价:79.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至zltts@phei.com.cn,盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线:(010) 88258888。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始,O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来,而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者,O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”;创建第一个商业网站(GNN);组织了影响深远的开放源代码峰会,以至于开源软件运动以此命名;创立了 Make 杂志,从而成为 DIY 革命的主要先锋;公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会聚集了众多超级极客和高瞻远瞩的商业领袖,共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择,O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程,每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列(真希望当初我也想到了)非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是一位特立独行的商人,他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了:‘如果你在路上遇到岔路口,走小路(岔路)。’回顾过去 Tim 似乎每一次都选择了小路,而且有几次都是一闪即逝的机会,尽管大路也不错。”

——Linux Journal

推荐序

“hypermedia as the engine of application state”

上面这段话看起来有些神秘，甚至不能算是一个完整的句子。它究竟是什么意思？它是一段咒语吗？它有什么神奇的魔力？

在我看来，如果把 Web 系统比作是电影《黑客帝国》(The Matrix) 里面那座精密宏伟、无与伦比的 Matrix 系统，上面这段话就是那位华人钥匙匠制作的、能够到达 Matrix 系统后台部分的钥匙。REST 之父 Roy Fielding 无疑是设计建造这座 Matrix 系统的主架构师之一，此外还有 Tim-Berners Lee 这样大神级的人物。这座 Matrix 系统的架构师不是一个人，而是一个英雄的团队。

Roy Fielding 博士一贯反对 design by buzzword (按照时髦的词汇来做设计)，在其 2000 年的博士论文中，他借用《建筑师讽刺剧》(The Architects Sketch) 里面某人所构思的一座到处悬挂着屠宰刀的公寓楼设计，来辛辣地讽刺那些只会 design by buzzword 的人。然而极为讽刺的是，Fielding 在其博士论文中创造出来的“REST” (包括后来出现的“RESTful API”) 这个缩写词现在已经成为了 Web 开发社区中最引人注目的一个 buzzword。很多人在尚未真正理解 REST 的一些核心概念的情况下，就到处公然宣称他们所设计的 API 是“RESTful API”。这样的情况太普遍了，以至于 Fielding 本人实在无法忍受，他在 2008 年 10 月写了一篇博客“REST APIs must be hypertext-driven” (RESTful API 必须是超文本驱动的)。这篇博客引起了 Web 开发社区广泛的讨论和反思，后来 Web 开发社区将 Fielding 在其博士论文中关于超媒体作用的论述，特别是本文开头的这句话，简化成了缩写词“HATEOAS”。“超文本驱动”和“HATEOAS”是完全相同的概念，只是表述方式不同，完全可以互换。

没错，HATEOAS 正是 REST 的灵魂，抛弃了 HATEOAS，REST 就失去了灵魂。虽然不支持 HATEOAS 的所谓“RESTful API”在很多场合仍然是很有用的，但是这样的 API 在松耦合和可伸缩性方面会受到一些损失。

Web 开发社区对于设计 RESTful API 最佳实践的探索，有点像 20 世纪初人类征服南极点或者征服更高山峰的竞赛，相关的图书反映出了竞赛的进展。从 2007 年第一本 REST 开发方面的图书《RESTful Web Services》，到今天为止这方面的图书已经不下 30 本。《RESTful Web APIs》这本新书，在对设计 RESTful API 的探索方面可谓是征服了一个新的高度。这本书并不是一本面向初学者的书，因此并没有重复其他 REST 开发入门图书中的一些基础知识的介绍，而是直接站在了前人的肩头（也包括三位作者本人以前写的两本 REST 开发图书《RESTful Web Services》和《Building Hypermedia APIs with HTML5 and Node》）。设计 RESTful API 的一些高级概念，例如对于超媒体的使用（HATEOAS）、各种为超媒体添加语义的技术、媒体类型的选择和设计、设计支持 HATEOAS 的 RESTful API 的流程等，在这本书里面讲的最为清楚。简单来说，这本书最核心的内容，就是如何设计支持 HATEOAS 的 RESTful API。

我在几年前翻译 Roy Fielding 博士论文的过程中，感觉论文虽然非常深刻、精彩，但是也比较抽象。REST 的这些核心概念，在具体的 RESTful API 设计和开发过程中如何实现，仍然是一个巨大的挑战。《RESTful Web APIs》这本书令人满意地解决了这些挑战，它把 REST 的抽象概念，忠实地具像化了，真的是非常棒的工作！

本书的中文版译者赵震一是我的一位朋友，我曾经和震一对于 REST 的一些核心概念做过深入探讨，震一对待技术问题的严谨态度让我印象非常深刻。震一和他的朋友李哲翻译的这本书，阅读起来非常流畅，体验很好，可以想见他们做过大量的推敲和润色。翻译技术图书，是一件辛苦的差事，需要大量的付出。我们在为原著作者的深厚功力而叹服的同时，也应该为好的译者而喝彩！

深入理解和学习 REST，就像是一种禅修的过程，可能需要持续几年时间。在这个过程中，有些时候会有顿悟，但是更多的还是渐悟。《RESTful Web APIs》这本书将会带领我们翻越一座新的山峰，过了那座山，后面还会有什么？

李锟

2014 年 5 月 22 日

对《RESTful Web APIs》一书的赞誉

“本书是学习 API 设计必备技艺的最佳起点。”

——Matt McLarty

API Academy 的联合创始人

“在阅读这本书的全部时间里，我一直在心里咒骂。我之所以这样做是因为在我逐个阅读了每一条解释之后，我不由地开始担心——它们写得实在太棒了，以至于在编写我自己的书时很难再找到一个更好的解释。你将再也无法找到另一部能将这个主题挖掘得如此彻底，解释得如此清晰的书了。请拿好这些工具，去创造一些奇妙的东西吧，并将它分享给世界上的其他人，好吗？”

——Steve Klabnik

《Designing Hypermedia APIs》一书的作者

“超媒体是最不容易被理解的 REST 要义，而本书对超媒体格式的解说精彩彻底。”

——Stefan Tilkov

REST 的布道者、作家及顾问

“超媒体 API 的最佳实用手册。人手必备。”

——Ruben Verborgh

语义超媒体研究员

献给 Sienna、Dalton 和 Maggie。——Leonard

献给“主管”Milo，不管是在本次还是很多其他的项目中，

你自始至终是我不变和耐心的朋友！——Mike

序

渐进呈现是用户界面设计中的一个概念，它提倡只在用户需要的时候呈现用户所需的信息。从很多方面来讲，你正在阅读的这本书就是一个实践了该原则的实例。而事实上，仅仅是回到七年前，这本书中所提到的内容很可能还无法“工作”。

正如你所见，相比于编写《RESTful Web Services》（本书的前身）之时，编程世界如今已时过境迁。在那个时候，“REST”这个词还是很少有人使用的。即便在有人使用时也往往被误用，人们对该词也存在着广泛的误解。

在 20 世纪 90 年代中后期，REST 所基于的标准 HTTP 和 HTML 已经被开发出来，并成为了 IETF 和 W3C 的标准，大致已接近它们现今的形态。尽管如此，还是存在着上述这样的情况。Roy Fielding 在他 2000 年发表的论文中引入了 REST 一词，而本书便是基于这篇论文所编写的。

Leonard Richardson 和我曾打算纠正这一（对 REST 来说）不公正的现状。为此，我们主要专注于那些支撑 HTTP 的基础概念，并且就如何将这些概念应用到实际应用中提供了一些具有实践意义的指导。

我认为我们的努力起到了松动这一现状根基的作用，并自此引发了对 REST 支持的雪崩效应。REST 迅速有了它自己的生命，并逐渐成为了一个流行的词汇。事实上现在的情况差不多是：只要有新的 web 接口推出，几乎默认都会称它为 REST。在短短的几年里，我们确实走了很长一段路。

不可否认，REST 作为一个词已经被用过了头，而且通常都没有被正确地应用。但是综合所有的考虑，我还是非常欣慰的，资源和 URI 的概念已经成功地渗入到应用接口设计之中。Web 毕竟是一个颇具弹性的地方，这些新的接口尽管不完美，但是比起那些它们所替代的老的接口却有着飞跃式的进步。

但是我们还可以做得更好。

如果将此事比喻成构建一座大厦，那么现在用于构建大厦的材料已经具备，是时候回过头来去重新审视一下整个领域，并基于这些概念来构建我们的大厦。下一步应该做的就是去探究那些通用的媒体类型以及特定的超媒体格式。上一本书几乎完全专注于构建正确的 HTTP 应用，而现在是时候让我们去深入探究下超媒体类型（那些像 HTML 一样没有与单独的应用或者甚至是单个厂商紧密绑定的媒体类型）背后的概念了。

HTML 就是这样一个超媒体格式的典型例子，它在 web 架构中占据着特殊的位置。事实上，我个人的发现之旅已经深入到了 HTML 的 W3C 标准的发展，也就是现在的 HTML5。虽然 HTML 确实在这本新书中占据了一个非常显著的位置，但是超媒体主题还有更多的内容需要讨论。所以尽管我们保持着联系，Leonard 选择了一个能够胜任我原来角色的人——Mike Amundsen 来作为本书的联合作者。

非常高兴能看到本书的编写，而在本书的阅读过程中我学习到了大量我从未在任何其他来源接触过的媒体类型。更重要的是，本书展示了这些类型所具有的共同点，以及如何对它们进行区分，因为它们中的每一个都具有自己的特性。

希望本书所做的一切能起到跟它的前身（《RESTful Web Services》）相同的效果。谁知道在另一个七年内所有的事情会不会有机会再重来一次呢，并且对表述性状态移交（Representational State Transfer）^{注1}中那些仍然被忽视的其他方面做出强调。

——Sam Ruby

注1 此处 Representational State Transfer 参考了李锐修订后的译法，译作“表述性状态移交”，此前曾被翻译为“表述性状态转移”——译者注。

前言

“大多数软件系统在创建时都有一个隐含的假设：整个系统处在一个实体的控制之下；或者至少参与到系统中的所有实体都向着一个共同目标行动，而不是有着各自不同的目标。当系统在互联网上开放地运行时，无法安全地满足这样的假设。”

——Roy Fielding

Architectural Styles and the Design of Network-based Software Architectures

“Discordia 信徒应该一直使用官方的 Discordian 文档编号系统。”

——Malaclypse the Younger 和 Lord Omar Khayyam Ravenhurst

Principia Discordia^{注1}

我要向你展示一种可以更好地进行分布式计算的方式，它使用了有史以来最成功的分布式系统，即万维网的根本思想。如果你已经决定（或者你的经理已经决定）需要为你的公司发布一个 web API 的话，我希望你能够读一下这本书。不管在你计划中的是一个公共的 API，还是一个纯粹的内部 API，抑或是一个只有受信伙伴可以访问的 API——它们都可以从 REST 的哲学中受益。

如果你想学习如何编写 API 客户端的话，那么这本书对你来说并不是必要的。这是因为大多数现有的 API 设计都基于一些有着数年之久的假设，而这些假设正是我想要摧毁的。

大部分今天的 API 都有着一个很大的问题：一旦部署，它们将无法改变。有一些大名鼎鼎的 API 会在一次部署后多年保持静态不变，即使围绕它们的行业发生着改变，这是因

注1 Discordia 是罗马神话中专司纷争与混乱的女神，Principia Discordia 则是一部与该女神相关的宗教书籍。——译者注

为要改变它们非常困难。

但是 RESTful 架构是为掌控变化而设计的。万维网由数百万的网站组成，运行在数千种不同的服务器实现之上，并且经历着周期性的重新设计。这些网站被数十亿的用户访问着，而这些用户使用着几十种硬件平台之上的数百种不同的客户端实现。你的部署一开始可能看上去不会如此混乱，但是当你的应用越发接近 web 的规模时，你将会看到越发相似的混乱景象。

要改变一个非常简单的系统通常都是很容易的。在规模很小时，一个 RESTful 系统比一个一键式的解决方案（push-button solution）需要花费更多预支的设计成本。但是当你的 API 逐渐成熟并开始发生变化时，你将会真正需要一些像 REST 这样的方式来应对变化。

- 一个商业上成功的 API 将保持连续多年的可用。一些 API 拥有数百甚至是数以千计的用户。就算问题域只是偶然地发生变化，对客户端带来的累积效应将是非常大的。
- 有一些 API 一直都在发生变化，新的数据元素和业务规则不断地被添加进来。
- 在某些 API 中，每个客户端都可以通过改变 workflow 来使其适合自己的需求。即使 API 自身从不变化，每个客户端对 API 的经历（鉴于经历不同的 workflow）将会不同。
- 编写 API 客户端的人通常不会和编写服务器的人隶属于同一个团队。所有向公共开放的 API 都属于这一类。

如果你不知道外部的客户端是哪种类型的话，你需要在做出变化时格外小心——否则你就需要一个能够在发生变化时保证不会破坏所有客户端的设计。如果你为你的 API 复制了现有的设计，你将很可能只是在重复以往犯过的错误。不幸的是，大部分的改进发生在幕后，它们大都还处于实验阶段并需要经过漫长的标准流程。我将会在本书中讨论到数十种特定的技术，包括很多还仍然处于开发之中。但是我的主要目标是要教会你 REST 的基本原则。通过对这些内容的学习，你将可以对任何实验成果以及那些通过流程审核的标准善加利用。

这里有两个我想在本书中尝试解决的具体问题：重复的工作以及对超媒体的逃避。让我们来看看它们。

重复的工作

现今已发布的 API 都是根据托管它们的公司的名字进行命名的。我们谈论着“Twitter API”、“Facebook API”和“Google+ API”。这三套 API 做着相似的事情。它们都拥有一些用户账户的概念，（在其他方面）它们都允许用户向自己的账户发布文本信息。但是每个 API 都具有完全不同的设计，学习一个 API 并不能帮助你学习下一个。当然，

Twitter、Facebook 和 Google 都是互相竞争的大型公司，它们并不想让你很容易地就会它们竞争对手的 API。但是小公司和非营利性组织也在做着相同的事。它们重新设计着自己的 API，就好比从来没有人在这方面有过相似的想法一样，但是这干扰了它们想让人们实际使用它们的 API 的目标。

让我来向你展示一个例子吧。网站 ProgrammableWeb (<http://www.programmableweb.com/>) 拥有着一个超过 8000 个 API 的目录。当我正在编写此书之时，它已经收录了 57 种微博 API——这些 API 的主要用途是向用户的账户发布文本信息^{注2}。很不错，有 57 家公司在这个领域发布了 API，但是我们真的需要 57 种不同的设计吗？我们在这里讨论并不是那些复杂难懂的业务，例如保险政策或法规守则，我们讨论的只是向用户账号发布少量的文本信息。你想成为那个设计第 58 种微博 API 的人吗？

最显而易见的解决方案便是创建一个微博 API 的标准。但是我们已经有了一个可以很好工作的标准：Atom 发布协议 (Atom Publishing Protocol)。它发布于 2005 年，然而几乎没有人使用它。有一些关于 API 的原因，使得每个人都想从头开始设计他们自己的 API，即使从业务的角度来看这并没有什么意义。

我不认为凭我一个人的力量就能结束这种无用功，但是我确实认为可以将问题分解成若干有意义的小块，然后提供一些方式来让新的 API 可以复用已经完成的这些工作。

超媒体很难

早在 2007 年，Leonard Richardson 和 Sam Ruby 编写了本书的前身，*RESTful Web Services* (O'Reilly)。那本书同样也尝试于解决两个大的问题。其中一个问题已经被解决，而另一个却没有任何进展^{注3}。

第一个问题是：在 2007 年，在 API 设计的多个阵营中，REST 学派正在与使用基于 SOAP 等重量级技术的对手学派进行对峙，他们忙于应对来自对方的对 REST 学派合理性的质疑。*RESTful Web Services* 一书的出现打破了这种对峙的僵局，有效地为 RESTful 设计原则防御了来自 SOAP 学派的进攻。

很好，这场对峙已经结束，而 REST 赢得了胜利。SOAP API 仍在被使用着，不过仅限于那些起初支持 SOAP 学派的大公司。几乎所有面向公众的 API 口头上都说自己遵守了 RESTful 原则^{注4}。

注2 具有微博(microblogging)标签的 ProgrammableWeb API 的完整列表提供了有关列表中每个 API 的信息。

注3 *RESTful Web Services* 一书现在是 O'Reilly 开放图书项目 (<http://oreilly.com/openbook/>) 中的一部分。你可以从该图书的页面下载到它的 PDF 版本。

注4 如果你想知道，这便是我们为什么更改了本书书名的原因。“web service”这个词与 SOAP 耦合得过于紧密，当 SOAP 没落之时，将会带着“web service”一词一起日暮西山。这些日子，每个人都开始改为谈论 API 了。

这又将我们带到了第二个问题：REST 并不只是一个技术词汇——它同样还是一个营销术语。在很长一段时间里，REST 成了一个口号，它象征着任何站在 SOAP 学派对立面的势力。任何没有使用 SOAP 的 API 都将自己标榜为 REST，即使它的设计与 REST 毫无关系甚至是违背了 REST 的基本原则。这样做是错误的，是令人困惑的，它给 REST 这个技术词汇带来了一个坏名声。

这种情况自 2007 年便有了较大的改善。每当我审视那些新的 API，我看到了开发者们的工作，可以看得出，这些开发人员是理解那些我将在本书前几章中解释的概念的。今天大部分举着 REST 大旗的开发者都理解资源和表述，理解如何使用 URL 来为资源命名，以及如何正确地使用 HTTP 方法。因此本书的前三章将不需要做过多的事情，只需让新的开发者加速赶上我们即可。

但是在 REST 中，还有一个方面令大部分的开发人员仍然无法理解，即：超媒体。我们都理解 Web 环境中的超媒体。它只是作为代表链接的一个华丽的词汇。网页经过互相的链接，随即产生了万维网，万维网便是由超媒体驱动的。但是，貌似只要在 web API 中涉及到超媒体，我们便有了心理障碍。这是一个大问题，因为超媒体是一项能让 web API 优雅处理变化的特性。

从 *RESTful Web APIs* 一书的第 4 章开始，我的首要目标便是教会你超媒体的工作原理。如果你从未听到过这个词，我将会结合其他重要的 REST 概念向你讲授该词。如果你听到过超媒体，但是这个概念吓到了你，我将会尽我所能来为你建立勇气。如果你无法将超媒体装进你的大脑，我将会以各种我所能想到的方式来向你展示超媒体，直到你记住并理解它。

RESTful Web Services 一书也涉及到了超媒体，但是这并不是该书的重心所在。就算跳过该书的超媒体部分也可以照样设计出一个功能性的 API。相比之下，*RESTful Web APIs* 则是一本真正有关超媒体的书。

我之所以这样做是因为超媒体是 REST 最重要的一个方面，也是最不被理解的一个方面。在我们完全理解超媒体之前，REST 将会被继续视为一个营销术语，而不是对处理分布式计算复杂性的一次认真的尝试。

这本书讲了什么？

前 4 个章节介绍了 REST 背后的概念，并将其应用于 web API。

第 1 章，网上冲浪

这一章通过一个你已经熟悉的 RESTful 系统：网站，来对基本的术语进行了说明。

第 2 章，一个简单的 API

这一章将我们在 Web 上的经验转换到了一个可编程 API 中，该 API 与第 1 章中所讨论的网站具有相同功能。

第 3 章，资源和表述

资源是 HTTP 中的基本概念，而表述则是 REST 中的基本概念。本章对它们之间是如何进行关联的进行了说明。

第 4 章，超媒体

超媒体是一种缺失的材料，可以将它和表述一起整合进一个一致的 API。本章展示了超媒体可以做些什么，并大都使用了你已经熟悉的超媒体数据格式：HTML。

接下去的 4 个章节描述了用于设计超媒体 API 的不同策略：

第 5 章，领域特定设计

显而易见的一个策略是设计一个全新的标准来处理你的具体问题。我使用了 Maze+XML 标准来举例说明。

第 6 章，集合模式（Collection Pattern）

很特别的一个模式：集合模式，在 API 设计中出现了一次又一次。在这一章中，我展示了两个不同的标准：Collection+JSON 和 AtomPub，它们都实现了这种模式。

第 7 章，纯 - 超媒体设计

当集合模式无法满足你的需求时，你可以使用一种通用的超媒体格式来传达任意的表述。这一章使用了 3 种通用超媒体格式（HTML、HAL 和 Siren）作为实例来展示上述方式是如何工作的。这一章同样还介绍了 HTML 微格式和微数据，并以此引出了下一章的内容。

第 8 章，Profile

Profile 可以用于填平某种数据格式（可以被多种不同的 API 使用）与某个特定 API 实现之间的鸿沟。我所推荐的 profile 格式是 ALPS，但是我同样也提到了 XMDP 和 JSON-LD。

在这一章中，我给出的建议开始超越了编写本书时的艺术状态（outstrip the state of the art）。因此我不得不为本书开发了 ALPS 格式，因为当时还没有其他可以完成这项工作的选择。如果你已经对基于超媒体的设计非常熟悉，那么你可以跳过前面的部分直接来到第 8 章，但是我不认为你应该跳过第 8 章。

第 9 章到第 13 章涉及到了例如选择正确的超媒体格式以及如何充分利用 HTTP 协议这些实用主题。

第 9 章, API 设计流程

这一章将本书到目前为止讨论过的所有内容整合在了一起, 并给出了一个用于设计 RESTful API 的按部就班的指引。

第 10 章, 超媒体动物园

为了展示超媒体的能力, 本章讨论了近 20 种标准化的超媒体数据格式, 它们中的大部分并没有在本书的其他章节中有所涉及。

第 11 章, API 中的 HTTP

本章给出了在 API 实现中使用 HTTP 的一些最佳实践。我同样也讨论了一些 HTTP 的扩展, 包括即将到来的 HTTP 2.0 协议。

第 12 章, 资源描述和 Linked Data

Linked Data 是语义网社区用以实现 REST 的方法。JSON-LD 毫无疑问是最重要的 Linked Data 标准。我们曾在第 8 章谈到过它, 而我在本章中对它进行了重温。本章同样讨论了 RDF 数据模型和一些我在第 10 章没能谈到的基于 RDF 的超媒体格式。

第 13 章, CoAP: 嵌入式系统的 REST

本章通过对 CoAP 的讨论结束了本书的核心部分, CoAP 是一个完全没有使用 HTTP 的 RESTful 协议。

附录 A, 状态法典

作为对第 11 章的扩展, 本附录对 HTTP 规范中定义的 41 个标准状态码以及一些作为扩展定义的有用的状态码进行了深入的考察。

附录 B, HTTP 报头法典

与附录 A 相似, 本附录也是对第 11 章的扩展。它为 HTTP 规范中定义的 46 个请求和响应报头, 以及一些扩展提供了详细的概述。

附录 C, 为 API 设计者准备的 Fielding 论文导读

本附录包括了一个围绕 REST 的基础文档 (即 Fielding 论文) 展开的深度讨论, 用以说明该文档在 API 设计中的意义。

词汇表

该词汇表包含了你在 *RESTful Web APIs* 一书中会经常遇到的一些术语的定义。如果

你想要熟悉基本概念，或者是需要一个快速的、用于浏览特定概念定义的提示工具，那么这将会是一个很好的去处。

这本书没讲什么

RESTful Web Services 是最早有关 REST 的一部书籍，并且涉及了很多的背景知识。幸运的是，现在已经有着超过一打的关于 REST 的各个方面的书，而这样便让 *RESTful Web APIs* 可以腾出精力来专注于核心的概念。

为了保持本书的专注度，我去除了部分你可能会希望我覆盖到的主题。我想要告诉你这本书没讲什么，这样你便可以选择不购买本书，从而不会为此感到失望：

- 本书没有涉及到客户端编程。编写客户端来消费基于超媒体的 API 是一种新的挑战。眼下，对一个通用的 API 客户端来说，我们可以拥有的就是一个能够发送 HTTP 请求的代码库。这在 2007 年如此，时至今日仍然如此。因为问题存在于服务器端。当你为一个现有的 API 编写客户端时，你将只能仰仗 API 设计者。我无法给你任何通用的建议，因为现在并不存在跨 API 的一致性。这就是为什么我在本书中鼓动大家保持对服务器端一致性的积极性的原因。当 API 之间变得更加相似，我们也就可以编写更为精细的客户端工具。
第 5 章包含了一些示例的客户端实现，并尝试对不同类型的客户端进行归类，但是如果你想要一本全部关于 API 客户端的书，那么这本书并不适合你。我不认为目前市场上存在着你想要的书。
- 世界上部署最广泛的 API 客户端应当属 JavaScript 的 XMLHttpRequest 代码库。在每一个浏览器中都拥有一份该代码的副本，而当今大部分的网站也都构建于那些专门设计供 XMLHttpRequest 消费的 API 之上。对于本书来讲，这个领域过于庞大而无法完全涉及。市场上有着全篇专门讲述 JavaScript 代码库的书籍。
- 我花费了相当多的时间来讨论 HTTP 的机制（第 11 章、附录 A 和附录 B），但是我没有覆盖到任何具有深度的特定 HTTP 主题，有一些主题，尤其是关于缓存和代理这样的 HTTP 中间组件的相关主题，我只是简单地在本书中有所涉及。
- *RESTful Web Services* 重点专注于将你的业务需求拆分成一组互相关联的资源。根据我 2007 年以来的经验，我深信将 API 设计作为一种资源设计来思考可以有效地避免考虑超媒体。但本书却采用了一种截然不同的方式，它专注于表述和状态转换，而非资源。
也就是说，资源设计的方式无疑也是有效的。如果想听听在该方向发展的建议，我推荐由 Subbu Allamaraju 编著的 *RESTful Web Services Cookbook*(O'Reilly)。

管理备注

本书有两名作者 (Leonard 和 Mike)，但是在编写本书的过程中，我们被合并成了一个第一人称，即“我”。

本书中的内容没有与任何特定的编程语言绑定。所有的代码都采用了在网络协议（通常是 HTTP）中发送的消息格式（通常是 JSON 或 XML 文档）。我将假定你已经熟悉了常规的编程概念，比如反模式和广度优先搜索，以及你对万维网是如何工作的有着一个基本的理解。

我将不会呈现真实的代码，但是我在第 1 章、第 2 章以及第 5 章中所讨论的服务器和客户端背后的真实代码是存在的。你可以通过 *RESTful Web APIs* 一书的 GitHub 仓库 (<https://github.com/organizations/RESTful-Web-APIs>)，或者是通过官方的网站 (<http://www.restfulwebapis.org/>) 来获取这些代码，从而可以自行将它运行起来。这些客户端和服务端是采用 JavaScript 编写的，使用的是 Node 代码库。

我之所以选择 Node 是因为它让我可以使用相同的编程语言来编写客户端及服务端的代码。你无须为理解某个客户 - 服务器事务的两端而需要费神地在两种编程语言之间来回切换。Node 是开源的，并且可以运行在 Windows、Mac 和 Linux 系统上。它很容易在这些操作系统上进行安装，而你应该无须碰触太多的麻烦就可以获取这些例子，并将它们运行起来。

我将这些代码托管在 GitHub 上，因为随着时间的变化，我可以非常容易地更新这些实现。这同样也使得读者们可以为这些示例的客户端和服务端贡献其他编程语言的移植版本。

理解标准

万维网并不是一个供科学研究的客观事物。它是一种社会建构 (social construct) —— 一组按特定方式做事的协议。幸运的是，它与其他的社会建构不同 (比如礼节)，Web 底层的协议通常是已经约定好的。人类的 web 底层最核心的协议就是 RFC 2616 (HTTP 标准)、HTML4 的 W3C 规范以及 ECMA-262 (JavaScript 的基础标准，也被称为 ECMAScript)。每个标准都做了不同的工作，在本书的课程中，我还将讨论数十个专门设计供 API 使用的其他标准。

这些标准的伟大之处在于它们给了你坚实的基础。你可以使用它们来构建一种全新的网站或 API，即某些没有人曾经尝试过的东西。并且无须向你的所有用户进行对整个系统的说明，你将只需要对那些新的部分进行说明即可。

而坏消息便是这些协议通常晦涩难读：采用拗口而精确的英文编写而成的、由 ASCII 文本组成的冗长的段落，像“should”这样的日常词汇也具有了技术含义，并被标准化为大写的“SHOULD”^{注 5}。市场上大量的技术书籍之所以热卖，便是因为很多人希望通过阅读书籍而避免直接阅读这样的标准文档。

好吧，我无法做出任何的担保。如果这些标准中的任何一个貌似会成为某个你可以在工作中采纳的选择，那么你必须愿意深入其规范并真正地去理解它（或者买一本能详细覆盖它的书籍）。对于像 Siren、CoAP 和 Hydra 这样的标准，我没有空间来为它们提供除基本概览之外的更多详情。更不用说一旦给出了太多的细节，将会让所有在工作中不需要这些特定标准的读者对此产生厌倦。

当你穿梭于标准的森林之中时，请记住一点，并不是所有的标准都具有相同的力量。有些标准制定得非常完善，每个人都在使用，如果你违背了这些标准，那么将会为你带来大量的麻烦。而其他标准只是某些人的一家之言，而他们想法也未必比你自己的想法更加高明。

我认为将标准划归到 4 个分类中将会很有帮助：fiat 标准、个人标准、公司标准以及开放标准。我将在本书中一直使用这些词，所以让我来逐个对它们进行深入的解释。

Fiat 标准

Fiat 标准并不是真正的标准；它们是一些行为。没有人认同它们。它们只是对某些人做事方式的一种描述。这些行为可以用文档记录下来，但是它缺少了作为标准的一个前提——其他人应该以相同的方式做事。

几乎当今的每个 API 都是一种 fiat 标准，即某种与特定公司相关联的一次性设计。这就是我们谈论着“Twitter API”、“Facebook API”以及“Google+ API”的原因。你可能需要通过理解这些设计来做好你的工作，并且可能要为这些设计编写你自己的客户端，除非你服务的便是我们讨论中的这家公司。请别指望在你自己的 API 中使用这样的设计。如果你复用了——一个 fiat 标准，我们不会说你的 API 符合某个标准，我们只会认为它是一种复制。

我在本书中尝试解决的一个主要的问题便是设计工作中数以百计的人年（person-year）都被束缚在制定 fiat 标准这种无法复用的工作中。必须要结束这样的事情。如今设计一个新的 API 意味着重新发明一长串轮子。一旦你的 API 完成了，你的客户端开发人员必须在客户端一侧对应地重新发明一串轮子。

注 5 “SHOULD” 的意见 RFC 2119。

即便在理想的情况下，你的 API 仍将是一个 fiat 标准，因为你的业务需求将会与其他人的需求有所不同。但是从观念上来说，一个 fiat 标准将只是一束掩盖了若干其他标准的强光。

当我在描述一个 fiat 标准时，我将会链接到它的人类可读的文档。

个人标准

个人标准是一种标准（你将被邀请阅读文档，并自行实现该标准），但是它们只是一个人的意见。我在第 5 章中描述的 Maze+XML 标准便是一个很好的例子。别指望 Maze+XML 是用来实现迷宫游戏 API 的标准方式，但是如果它能为你工作，你便可以很好地使用它。某些其他的人已经帮你完成了设计工作。

个人标准相对于别的标准，通常会采用较为不够正式的语言。很多开放标准开始是作为个人标准起步的——在经历了大量的实验后，作为编外项目被扶正。我在第 7 章中谈到 Siren 就是一个很好的例子。

当我在描述一个个人标准时，我将会链接到它的规范。

公司标准

公司标准是由企业集团为了尝试解决某个困扰着它们的问题而共同创建的标准，或者是由一个单一的公司试图代表其客户解决某个反复出现的问题而制定的。公司标准相比于个人标准往往定义得更加完善，所使用的语言也更加规范，但是它们并没有比个人标准具有更大的力量。它们只是某个公司（或某个企业集团）的意见。

公司标准包括了活动流（Activity Streams）和 schema.org 的元数据模式，这两项都在第 10 章有所涉及。很多的行业标准都是以公司标准的形式发起的。OData（同样在第 10 章中讨论过）最初以微软项目的形式启动，但是在 2012 年被提交到了 OASIS，并最终成为了一项 OASIS 标准。

当我在描述一个公司标准的时候，我将会链接到它的规范。

开放标准

一项开放标准将会历经一个由委员会发起的设计过程，或者至少拥有一个开放的评论期，在此期间，大量的人员会阅读规范，对它提出意见，并提供改进的建议。在这个过程的最后，该规范将得到某些公认的标准组织的祝福。

该过程给开放标准带来了一定的道德力量。如果一个开放标准的内容或多或少正是你想

要的，你真应该使用它来替代制定自有的 fiat 标准。开放标准在设计过程和评论期间可能会暴露出大量的问题，这样一来你在实际的使用中将不大会遇到太多的问题。

一般来说，伴随着开放标准还会有很多的协议，这些协议会对你做出承诺，即当你在实现了这些开放标准之后不会受到那些参与了该标准过程的公司的专利侵权诉讼。相比之下，实现别人的 fiat 标准有可能会激起他们对你的专利侵权诉讼。

本书中提到的一些开放标准大都出自那些大名鼎鼎的标准组织：ANSI、ECMA、ISO、OASIS，尤其是 W3C。我无法描述在成为这些标准组织的一员后将可以怎么样，因为我从未这样做过。但是大部分重要的标准组织^{注 6}中的任何一个都可以向 IETF 做出贡献，该组织管理着所有重要的 RFC。

RFCs(Requests for Comments) 和互联网草案

大部分的 RFC 都是通过一个叫作标准追踪（Standards Track）的流程进行创建的。贯穿本书，我将会引用那些处于标准追踪不同环节的文档。我想要简明地讨论下追踪是如何运作的，这样一来你便会知道在采纳我的建议时需要多么认真。

一项 RFC 的生命开始于一项互联网草案。这是一个看上去很像标准文档的文档，但是你不应该建立基于它的实现。你应该查找该规范的问题并给予反馈。

一项互联网草案具有一个固定 6 个月的生命时间。在它发布的 6 个月之后，该草案必须作为一项 RFC 被通过，或者是由一项更新后的草案进行替代。如果以上的事情都没有发生，那么该草案就会被认为已经过期，并且不应该被应用于任何事物。另一方面，如果草案通过了审核，那么它就会立即过期并由一项 RFC 来取代它。

因为具有固定的过期时间，同时也因为一项互联网草案从技术上说并不是任何类型的标准，所以在一本书中提及它们是一件非常棘手的事情。同时，API 设计是一个迅速变化的领域，多一项互联网草案可以选择总比没有好。我将在本书中提到多个互联网草案，并且假设它们在成为 RFC 后不会有太大的变化。这个假设保持着良好的状态；有多个在我编写本书时提到的互联网草案，目前已经成为了 RFC。如果某个特定的互联网草案最后并不成功，那么我在此只能先行道歉了。

RFC 和互联网草案都指定了代码名称。当我描述它们中的一员时，我不会链接到它的规范。我将只会提及它的代码并让你自己去查找它。举个例子，我会将 HTTP/1.1 规范称为 RFC 2616。我也会使用名字来指代某个互联网草案。举个例子，我将会使用“draft-snell-link-method”来指代向 HTTP 添加 LINK 和 UNLINK 方法的提案。

注 6 不管怎样，我们要达成这本书的目的。如果你需要螺丝和螺栓的标准型号，你将需要 ANSI 或 ISO。

当你看到它们中某个的代码名称时，你都可以通过网络搜索来找到该 RFC 或互联网草案的最新版本。如果一项互联网草案在本书出版之后成为了 RFC，那么该互联网草案的最终版本会链接到对应的 RFC。

当我在描述一项 W3C 或 OASIS 标准时，我将会链接到对应的规范，因为那些规范没有给出代码名称。

本书所使用的约定

以下是本书所使用的排版约定：

斜体

表示新的术语（中文则用楷体）、URL、邮件地址、文件名称和文件扩展名。

等宽字体

用于表示程序片段，也可以用在正文中表示例如变量名或函数名等程序元素、数据库、数据类型、环境变量、语句和关键词（中文则用黑体）。

加粗的等宽字体

展示了应该由用户逐字输入的命令或其他文本。

倾斜的等宽字体

展示了应该由用户提供值或由上下文决定值来进行替换的文本。



这个图标代表小窍门、建议和说明。



这个图标代表警告信息。

使用代码示例

本书就是要帮读者完成工作的。通常，如果本书包含了代码示例，你可以在你的程序和文档中使用本书中的代码。除非你复制了大段的代码，否则你无须联系我们来取得许可。举个例子，在编写程序时使用了本书中的数块代码是不需要经过许可的。出售或分发来自 O'Reilly 图书的示例 CD-ROM 是必须经过许可的。引用本书及本书中的示例代码来回答问题是不需要经过许可的。将大量的示例代码整合到你的产品文档中必须经过许可。

我们希望（但不是必须）你在使用我们的代码时标明出处。出处通常都包含书名、作者、

出版社和 ISBN。例如：“*RESTful Web APIs* by Leonard Richardson and Mike Amundsen (O'Reilly). Copyright 2013 Leonard Richardson and amundsen.com, Inc., and Sam Ruby. 978-1-449-35806-8”。

如果还有其他需要使用代码的情形需要与我们沟通，可以随时与我们联系：
permissions@oreilly.com。

Safari® Books Online

Safari Books Online (www.safaribooksonline.com) 是一家应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，请访问我们的网站。

联系我们

关于本书的建议和疑问，可以与下面的出版社联系：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室 (100035)
奥莱利技术咨询(北京)有限公司

我们将关于本书的勘误表，例子以及其他信息列在本书的网页上，网页地址是：

<http://www.oreilly.com/catalog/9781449358068>

如果要评论本书或者咨询关于本书的技术问题，请发邮件到：

bookquestions@oreilly.com

想了解关于 O'Reilly 图书、课程、会议和新闻的更多信息，请访问以下网站：

<http://www.oreilly.com.cn>

<http://www.oreilly.com>

致谢

我们欠 Glenn Block 一声谢谢，他花费了无数的时间来聆听我们的想法，并编写了真实的代码来测试这些想法。我们要向 Benjamin Young 和 RESTFest 的所有人表示感谢，是他们同意成为我们实验的一部分，并给了我们很好的反馈和建议，即使我们有时听不大进去。我们要感谢 Mike 在 Layer 7 Technologies 的同事，包括 Dimitri Sirota 和 Matt McLarty，他们在这个项目的工作上给予了他支持和鼓励。我们要感谢 Sam Ruby 和 Mike Loukides，他们对于本书的前身 *RESTful Web Services* 一书来说至关重要。我们要感谢 Sumana Harihareswara，她是 Leonard 的贤内助。我们还要感谢社区，它为我们创造了良好的协作和交流 REST 与 API 的环境；尤其是 Yahoo 的 REST-Discuss、Google Groups 的 API-Craft 以及 LibreList 的 Hypermedia group。

最后，感谢那些阅读了早期手稿并提供了必要的批评和支持的人，他们是：Carsten Bormann、Todd Brackley、Tom Christie、Timothy Haas、Jamie Hodge、Alex James、David Jones、Markus Lanthaler、Even Maler、Mark Nottingham、Cheryl Phair、Sergey Shishkin、Brian Sletten、Mark Stafford、Stefan Tilkov、Denny Vrandečić、Ruben Verborgh 和 Andrew Wahbe。

目录

序	xix
前言	xxi
第 1 章 网上冲浪	1
场景 1：广告牌	2
资源和表述	2
可寻址性	3
场景 2：主页	3
短会话（Short Session）	5
自描述消息（self-descriptive message）	5
场景 3：链接	6
标准方法	8
场景 4：表单和重定向	9
应用状态（Application State）	11
资源状态（resource state）	12
连通性（connectedness）	13
与众不同的 Web	14
Web API 落后于 Web	15
语义挑战	16

第 2 章 一个简单的 API	17
HTTP GET : 安全的投注	18
如何读取 HTTP 响应	19
JSON	20
Collection+JSON	21
向 API 写入数据	23
HTTP POST: 资源是如何生成的	24
由约束带来解放	26
应用语义所产生的语义鸿沟	27
 第 3 章 资源和表述	 29
万物皆可为资源	30
表述描述资源状态	30
往来穿梭的表述	31
资源有多重表述	32
HTTP 协议语义 (Protocol Semantics)	33
GET	35
DELETE	36
幂等性 (Idempotence)	36
POST-to-Append	37
PUT	38
PATCH	39
LINK 和 UNLINK	40
HEAD	40
OPTIONS	41
Overloaded POST	41
应该使用哪些方法?	42
 第 4 章 超媒体	 45
将 HTML 作为超媒体格式	46
URI 模板	49
URI vs URL	50
Link 报头	51

超媒体的作用	52
引导请求	52
对响应做出承诺	54
工作流控制	55
当心冒牌的超媒体!	56
语义挑战：我们该怎么做?	57
第 5 章 领域特定设计	59
Maze+XML：领域特定设计	60
Maze+XML 是如何工作的	61
链接关系	62
访问链接来改变应用状态	64
迷宫集合	65
Maze+XML 是 API 吗?	67
客户端 1：游戏	68
Maze+XML 服务器	72
客户端 2：地图生成器	74
客户端 3：吹牛者	76
客户端做自己想要做的事	77
对标准进行扩展	77
地图生成器的缺陷	80
修复（以及修复后的瑕疵）	81
迷宫的暗喻	83
解决语义鸿沟	83
领域特定设计在哪里?	83
最终的奖赏	84
报头中的超媒体	84
抄袭应用语义	84
如果找不到相关的领域特定设计，不要自己制造	86
API 客户端的种类	86
人类驱动的客户终端	86
自动化客户终端	87

第 6 章 集合模式 (Collection Pattern)	91
什么是集合?	93
链向子项的集合	93
Collection+JSON	94
子项的表示	95
写入模板 (Write Template)	98
搜索模板	99
一个 (通用的) 集合是如何工作的	100
GET	101
POST-to-Append	101
PUT 和 PATCH	101
DELETE	102
分页	102
搜索表单	103
Atom 发布协议 (AtomPub)	103
AtomPub 插件标准	105
为什么不是每个人都选择使用 AtomPub?	106
语义挑战: 我们应该怎么做?	107
第 7 章 纯 - 超媒体设计	111
为什么是 HTML?	111
HTML 的能力	112
超媒体控件	112
应用语义插件	113
微格式	115
hMaze 微格式	116
微数据	118
改变资源状态	119
为表单添加应用语义	121
与超媒体相对是普通媒体	125
HTML 的局限性	126
拯救者 HTML5?	127
超文本应用语言	128

Siren.....	131
语义挑战：我们现在要怎么做？	133
第 8 章 Profile	135
客户端如何找寻文档？	136
什么是 Profile？	137
链接到 Profile	137
Profile 链接关系	137
Profile 媒体类型参数	138
特殊用途的超媒体控件	139
Profile 对协议语义的描述.....	139
Profile 对应用语义的描述.....	140
链接关系	141
不安全的链接关系	142
语义描述符	142
XMDP：首个机器可读的 Profile 格式	143
ALPS.....	146
ALPS 的优势.....	150
ALPS 并不是万金油.....	152
JSON-LD	153
内嵌的文档	156
总结	158
第 9 章 API 设计流程	161
两个步骤的设计流程	161
七步骤设计流程.....	162
第 1 步：罗列语义描述符	163
第 2 步：画状态图	164
第 3 步：调整命名	168
第 4 步：选择一种媒体类型	172
第 5 步：编写 Profile	173
第 6 步：实现.....	174
第 7 步：发布.....	174

实例：You Type It, We Post It.....	177
罗列语义描述符	177
画状态图	178
调整名称	179
选择一种媒体类型	180
编写 Profile	181
设计建议	182
资源是实现的内部细节	182
不要掉入集合陷阱	183
不要从表述格式着手	184
URL 设计并不重要	184
标准名称优于自定义名称	186
设计媒体类型	187
当你的 API 改变时	189
为现有 API 添加超媒体	194
改进基于 XML 的 API	195
值不值得?	196
Alice 的第二次探险	196
场景 1：没有意义的表述	196
场景 2：Profile	198
Alice 明白了	200

第 10 章 超媒体动物园 203

领域特定格式	204
Maze+XML	204
OpenSearch	205
问题细节文档	205
SVG	206
VoiceXML	208
集合模式的格式	210
Collection+JSON	211
Atom 发布协议	211
OData	212

纯超媒体格式	219
HTML.....	219
HAL	220
Link 报头	222
Location 和 Content-Location 报头	222
URL 列表	223
JSON 主文档 (Home Documents)	223
Link-Template 报头	224
WADL	225
XLink	226
XForms.....	227
GeoJSON : 一个令人困惑的类型	228
GeoJSON 没有通用的超媒体控件.....	230
GeoJSON 没有媒体类型	232
从 GeoJSON 学习到的经验.....	233
语义动物园	234
链接关系的 IANA 注册表	234
微格式 WiKi.....	235
来自微格式 Wiki 的链接关系.....	236

第 11 章 API 中的 HTTP 241

新 HTTP/1.1 规范	242
响应码.....	242
报头	243
表述选择	243
内容协商 (Content Negotiation)	243
超媒体菜单	244
标准 URL (Canonical URL)	245
HTTP 性能	246
缓存 (Caching)	246
条件 GET 请求 (Conditional GET)	247
Look-Before-You-Leap 请求.....	249
压缩.....	250

部分 GET 请求 (Partial GET)	250
Pipelining	251
避免更新丢失问题	252
认证	254
WWW-Authenticate 报头和 Authorization 报头	255
Basic 认证	255
OAuth 1.0	256
OAuth 1.0 的缺点	259
OAuth 2.0	260
何时不采用 OAuth	261
HTTP 扩展	261
PATCH 方法	262
LINK 和 UNLINK 方法	262
WebDAV	263
HTTP 2.0	264

第 12 章 资源描述和 Linked Data 267

RDF	268
RDF 将 URL 作为 URI 对待	270
什么时候使用描述策略	271
资源类型	273
RDF Schema	274
Linked Data 运动	277
JSON-LD	278
将 JSON-LD 作为一种表述格式	279
Hydra	280
XRD 家族	285
XRD 和 JRD	285
Web 主机元数据文档	286
WebFinger	287
本体动物园 (Ontology Zoo)	289
schema.org RDF	289
FOAF	290

vocab.org	290
总结：描述策略生机盎然！	290
第 13 章 CoAP: 嵌入式系统的 REST	293
CoAP 请求	294
CoAP 响应	294
消息种类	295
延迟响应 (Delayed Response)	296
多播消息 (Multicast Message)	296
CoRE Link Format	297
结论：非 HTTP 协议的 REST	298
附录 A 状态法典	301
附录 B HTTP 报头法典	325
附录 C 为 API 设计者准备的 Fielding 论文导读	349
词汇表	365

网上冲浪

万维网变得越来越普遍的一个重要原因就是它的易用性，普通人不需要经过多少培训就可以使用它来完成一些很有价值的工作。但是在这表象之后，Web 也是一个用于分布式计算的功能强大的平台。

那些让普通人易于使用互联网的原则，也同样适用于某些自动化的软件 agent “用户”。比如，一些软件被设计用来在不同的银行账户间自动地进行货币交易（或者用来完成现实世界的其他任务），为了完成相应的任务，它们所采用的基本技术和人类所使用的技术是相同的。

就本书而言，有三项技术支撑着当今的互联网，它们分别是：URL 命名约定、HTTP 协议和 HTML 文档格式。URL 和 HTTP 是比较简单的，但是如果想要将它们应用到分布式编程中，和大多数的 web 开发人员相比，你就必须更加了解它们的细节。本书的前几章就专门用来帮助大家学习这些内容。

HTML 的内容就有点复杂了。在 web API 的世界里，有许许多多的数据格式在试图取代 HTML 的地位。对这些数据格式的研究将从第 5 章开始，并占据本书的多个章节的篇幅。暂时，我将重点放在 URL 和 HTTP 上面，仅仅使用 HTML 作为一个例子。

我计划从一个和万维网相关的简单故事开始，以此来解释隐藏在万维网设计背后的原则以及驱动它成功的缘由。虽然你对 Web 很熟悉，但是对于那些使得 Web 运转起来的技术和概念你可能并没听说过，所以这个故事是很简单的，以便于你能够理解它。如果你对类似于“将超媒体作为应用状态的引擎 (hypermedia as the engine of application state)”的技术不是很清楚，我希望你能通过这个简单而又具体的例子来得到一些收获。

那我们现在开始吧。

2 场景1：广告牌

一天，Alice 正在城里散步，她看到了一个广告牌（见图 1-1）。



图1-1 广告牌

（顺便说一下，这个虚构的广告牌上面宣传的是我为这本书设计的一个真实的网站，你可以自己试着访问一下。）

这个广告牌是在 20 世纪 90 年代中期建立的，那时候 Alice 已经可以记事了，她至今都还记得这个广告牌刚开始展示这个 URL 时候公众的反应。起初，人们都拿这些看起来很怪异的字符串开玩笑，那时的人们并不清楚“http://”和“youtypeitwepostit.com”的真正含义。但是 20 年后的今天，几乎每个人都知道如何使用这个 URL：将 URL 输入到 web 浏览器的地址栏，然后按下回车键。

Alice 也是这样做的：她拿出了自己的手机，将“<http://www.youtypeitwepostit.com/>”输进了浏览器地址栏。我们的故事的第一个场景就到此结束了，我们保留一个悬念：URL 的另一端是什么呢？

资源和表述

非常抱歉中断了这个故事，但是我需要介绍一些基本术语。Alice 的 web 浏览器会试图向特定的 web 服务器的 URL:<http://www.youtypeitwepostit.com/> 发送一条 HTTP 请求。一个 web 服务器可以管理许多不同的 URL，并且每个 URL 被授权访问服务器上的不同数据。

我们所说的 URL 是一些事物的 URL，比如一个产品、一个用户或主页等。这些用 URL

命名的事物的专业术语是资源 (resource)。

这个 URL:<http://www.youtypeitwepostit.com/> 标识的资源应该就是那个广告牌上宣传的网站的主页。但是也许只有将这个故事继续下去，等到 Alice 的浏览器真正发送了 HTTP 请求以后，我们才能确定这些猜想。

web 浏览器为一个资源发送了 HTTP 请求以后，服务器会发送一个文档作为响应（通常是一个 HTML 文档，但是有时候是二进制图片或者其他东西）。不论服务器发送了什么文档，我们都将这个文档称为资源的表述 (representation of the resource)。

◀ 3

每个 URL 标识一个资源。客户端向某个 URL 发送了一条 HTTP 请求以后，它就会收到对应资源的表述。客户端从来都不会直接看到资源，看到的都是资源的表述。

我会在第 3 章讲解更多关于资源和表述的内容。现在我只是想要使用资源和表述这两个术语来开始讨论可寻址性原则。

可寻址性

每个 URL 代表一个也仅代表一个资源。如果一个网站上面有两个从概念上讲并不相同的事物，那么我们就应该将它们作为两个资源并为它们分配不同的 URL。当网站违背了这个规则的时候，我们便会因为在使用时无所适从而心情沮丧。有一些餐厅的网站在这一点上做得就很差劲。有时候，整个网站堆砌在一个 Flash 界面里，并没有提供那些指向我们就其本身可以讨论的资源的独立 URL，比如菜单以及用来显示餐厅地址的地图等。

可寻址性原则就是说每个资源应该有一个属于自己的 URL。如果你的程序里面有些东西非常重要，它就应该有一个唯一的名字，一个 URL，这样你和你的用户就可以非常清楚地、毫无歧义地引用它了。

场景2：主页

回到我们的故事中，当 Alice 将广告牌上的 URL 输入到她的浏览器地址栏的时候，她就通过因特网向 web 服务器上面的 <http://www.youtypeitwepostit.com/> 发送了一个 HTTP 请求：

```
GET / HTTP/1.1
Host: www.youtypeitwepostit.com
```

web 服务器处理这个请求（Alice 和她的 web 浏览器都不需要知道是如何处理的）然后发送响应结果：

```
HTTP/1.1 200 OK
Content-type: text/html
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Home</title>
  </head>
  <body>
    <div>
      <h1>You type it, we post it!</h1>
      <p>Exciting! Amazing!</p>

      <p class="links">
        <a href="/messages">Get started</a>
        <a href="/about">About this site</a>
      </p>
    </div>
  </body>
</html>
```

4

在响应信息的头部的 200 是状态码，也被叫作响应码。这是服务器告诉客户端发生了什么事情的快捷方式。实际上还有很多其他的 HTTP 状态码，我会在附录 A 中对它们都进行一一介绍，但是最常见的状态码是我们前面见到的状态码 200。200 (OK) 表示这个请求被接受并正确无误地处理了。

Alice 的 web 浏览器将响应数据作为 HTML 文档进行了解析，并以图形化的形式展示了出来（见图 1-2）。



图1-2 You Type It... 主页

现在 Alice 可以浏览这个网页了，她终于明白那个广告牌在说什么了。这个广告牌是在给一个类似于 Twitter 的微博网站做宣传，尽管这个网站实际上并不像广告牌说的那样令人兴奋，但是作为一个例子也已经足够了。

Alice 和 web 服务器的第一次即时互动揭示了 Web 的一系列更重要的特性。

短会话（Short Session）

故事发展到了这里，Alice 的 web 浏览器正在显示那个网站的主页。从她的角度来看，她已经“登录到”了这个页面，这也就是她在虚拟的网络空间的当前位置。但是就服务器考虑而言，Alice 哪也不存在。服务器也已经忘记了她的存在。

HTTP 会话只维持在一次请求过程中，客户端发送请求，服务器进行响应。这意味着，Alice 可以将她的手机关一晚上，然后，当她的浏览器从内部缓存（cache）中恢复出这个网页以后，她依旧可以单击页面上的任意一个链接，网页应该还是可以继续工作的（和 SSH 会话相比，如果你将你的电脑关机，SSH 会话就终止了）。

Alice 甚至可以将她手机里面的网页一直打开着，她在六个月后再次单击网页上面的链接时，web 服务器还是会迅速地做出响应，仿佛她只是等待了几秒钟。Web 服务器不需要为了 Alice 而通宵工作。当 Alice 不发送 HTTP 请求的时候，服务器并不清楚 Alice 的存在。

这项原则有时候就被称为无状态性（statelessness）。我想这是一个令人困惑的术语，因为系统里面的客户端和服务端都要保存状态：它们只是保存不同类型的状态。“无状态性”术语是指服务器不关心客户端的状态（在后面的章节中，我将讲解不同类型的状态）。

自描述消息（self-descriptive message）

通过查看前面的 HTML，我们会清楚看到这个网站不仅仅有一个主页。主页的标签包含两个链接：其中一个是相对路径 URL “/about”（也就是 <http://www.youtypeitwepostit.com/about>），另一个是“/messages”（也就是 <http://www.youtypeitwepostit.com/messages>）。起初，Alice 只知道一个 URL——那个 URL 指向主页——但是现在她知道三个 URL 了。服务器正在慢慢将它的结构揭示给 Alice。

我们现在可以根据服务器揭示给 Alice 的信息给网站画一张地图了（见图 1-3）。

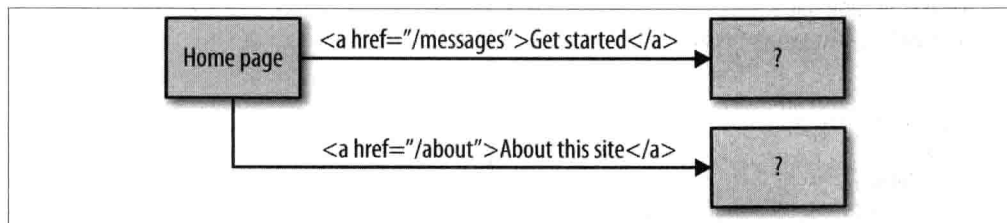


图1-3 网站地图

/messages 和 */about* 这两个链接的背后是什么呢？唯一确定的办法就是单击这些链接并找到真相。但是在这之前，Alice 可以查看一下 HTML 标记或浏览器呈现的可视化页面，进而根据这些信息来做一下推测：写有“About this site”文字的链接很可能是指向一个介绍这个网站的网页，而另一个写着“Get started”的链接很可能是能够让她真真正正地发布一条消息的页面。

6 当你请求一个网页的时候，你收到的 HTML 文档不仅仅可以提供给你所要求的即时信息，还会帮助你来决定下一步的操作。

场景3：链接

在浏览了主页以后，Alice 决定继续对这个网站做出进一步的尝试。她很自然地单击了那个写着“Get started”的链接。任何时候，你在浏览器上单击一个链接都表示你要求浏览器发送一个 HTTP 请求。

Alice 所单击的那个链接的代码如下：

```
<a href="/messages">Get started</a>
```

她的浏览器和上次一样向同样的服务器发送了一个 HTTP 请求：

```
GET /messages HTTP/1.1  
Host: www.youtypeitwepostit.com
```

请求中的 GET 是一个 HTTP 方法（HTTP method），也就是大家所知道的 HTTP 动词（HTTP verb）。HTTP 方法是客户端告诉服务器端它想要如何操作一个资源的方法。“GET”方法是最常见的 HTTP 方法。它的意思是“将这个资源的表述提供给我”。对于浏览器而言，GET 是默认值。当你点击一个链接，或者向地址栏输入一个 URL 的时候，你的浏览器就发送了一个 GET 的请求。

服务器处理这个特定的 GET 请求，发送了 */messages* 的表述给浏览器：

```
HTTP/1.1 200 OK  
Content-type: text/html  
...  
  
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Messages</title>  
  </head>  
  <body>  
    <div>
```

```

<h1>Messages</h1>

<p>
  Enter your message below:
</p>

<form action="http://youtypeitwepostit.com/messages" method="post">
  <input type="text" name="message" value="" required="true"
    maxlength="6"/>
  <input type="submit" value="Post" />
</form>

<div>
  <p>
    Here are some other messages, too:
  </p>
  <ul>
    <li><a href="/messages/32740753167308867">Later</a></li>
    <li><a href="/messages/7534227794967592">Hello</a></li>
  </ul>
</div>

<p class="links">
  <a href="http://youtypeitwepostit.com/">Home</a>
</p>

</div>
</body>
</html>

```

7

像之前一样，Alice 的浏览器将 HTML 文本进行了图形渲染（见图 1-4）。



图1-4 You Type It… “Get started” 页面

Alice 浏览这个页面以后，她会发现这个页面是一个消息列表，列表里面罗列了其他人已经发布到这个网站的消息，页面的上方还有一个引人注目的文本框和一个 Post 按钮。

现在我们获取到了这个服务器运行的更多信息，图 1-5 展示了到现在为止，Alice 的浏览器所了解到的新的网站地图。

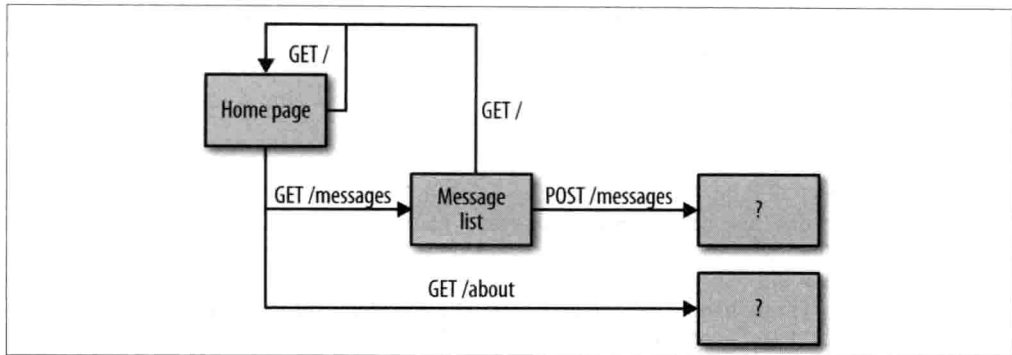


图1-5 关于You Type It...网站的浏览器的视图

标准方法

Alice 的浏览器前两次发送的 HTTP 请求都是使用 GET 作为它们的 HTTP 方法，但是在这个最新的表述里面却有一部分特殊的 HTML，它们的作用是当 Alice 单击 Post 按钮时触发一个 HTTP POST 请求：

```

<form action="http://youtypeitwepostit.com/messages" method="post">
  <input type="text" name="message" value="" required="true"
    maxlength="6"/>
  <input type="submit" />
</form>

```

HTTP 标准 (RFC 2616) 定义了客户端可以应用到一个资源上的 8 种方法。在本书中，我将重点放在其中的 5 个方法上面，它们分别是 GET、HEAD、POST、PUT 和 DELETE。在第 3 章中，我会对这些方法以及一个扩展方法 PATCH 进行更加细致的讲解。现在，最重要的事情就是记住这里有一些标准方法。

提出新的 HTTP 方法也不是不可能的 (PATCH 方法就是后来提出的)，但是要付出很大的代价。在这个方面，它并不像一门编程语言：在编程语言的世界里，你可以将你的方法命名为任何东西。在我为这个例子而搭建这个简单的微博网站的时候，我并没有定义类似于 GETHOME PAGE (获取主页) 和 HELLOPLEASESHOWMETHEMESSAGELIST THANKSBYE (你好，请向我显示消息列表，谢谢，再见) 这样的新 HTTP 方法。我使

用 GET 方法来同时实现了“显示主页”和“显示消息列表”的目的，因为在这两个例子中，GET 方法（“将资源的表述提供给我”）最符合 HTTP 接口以及我想要的。我并不是通过定义新的方法来区分主页和消息列表页，而是把这两个文档看作不同的资源：每个资源都有它自己的 URL，每个资源都可以通过 GET 进行访问。

场景4：表单和重定向

回到我们的故事，Alice 对这个微博网站的表单很有兴趣，她在文本框中输入了“Test”，然后单击了 Post 按钮。

现在，Alice 的浏览器又发送了一个 HTTP 请求：

```
POST /messages HTTP/1.1
Host: www.youtypeitwepostit.com
Content-type: application/x-www-form-urlencoded

message=Test&submit=Post
```

服务器返回的信息如下：

```
HTTP/1.1 303 See Other
Content-type: text/html

Location: http://www.youtypeitwepostit.com/messages/5266722824890167
```

Alice 的浏览器之前发送的两个 GET 请求，服务器返回了 HTTP 状态码 200（“OK”），并提供了一个可用于浏览器显示的 HTML 文档。但是这一次，服务器并没有返回 HTML 文档，取而代之的是在 Location 报头里面提供了另一个 URL 链接以及在响应信息头部提供状态码 303（“See Other”），而不是 200（“OK”）。

状态码 303 是告诉 Alice 的浏览器要自动向 Location 报头提供的那个 URL 发起第四个 HTTP 请求。这个过程不需要获得 Alice 的许可，浏览器仅仅执行了下面的操作：

```
GET /messages/5266722824890167 HTTP/1.1
```

这一次，浏览器返回了 200（“OK”）状态码以及一个 HTML 文档：

```
HTTP/1.1 200 OK
Content-type: text/html

<!DOCTYPE html>
<html>
  <head>
    <title>Message</title>
  </head>
```



```

<body>
  <div>
    <h2>Message</h2>
    <dl>
      <dt>ID</dt><dd>2181852539069950</dd>
      <dt>DATE</dt><dd>2014-03-28T21:51:08Z</dd>
      <dt>MSG</dt><dd>Test</dd>
    </dl>
    <p class="links">
      <a href="http://www.youtypeitwepostit.com/">Home</a>
    </p>
  </div>
</body>
</html>

```

10

Alice 的浏览器以可视化的方式显示了这个文档(见图 1-6),然后就继续等待 Alice 的输入。



图1-6 You Type It... 所提交的消息



我可以肯定你之前一定遇到过 HTTP 重定向, HTTP 协议就包含了很多类似于 HTTP 重定向的细微的功能特性, 其中有些可能你都没有接触过。我们有很多方法来让服务器告诉客户端如何采用不同的方式来处理一个响应, 我们也有很多方法来让客户端将条件或者额外的功能特性添加到一个请求中。API 设计的很大一部分工作内容就是恰当地使用这些功能特性。第 11 章介绍了 HTTP 中对于 web API 非常重要的一些功能特性, 并且附录 A 和附录 B 提供了关于这一主题的补充信息。

通过浏览图形化页面, Alice 看到她提交的消息 (“Test”) 现在已经是 YouTypeItWePostIt.com 网站上面一个正式的帖子了。我们的故事到此就结束了。Alice 已经完成了她的试用这个微博网站的目标, 但是在这四次简单的交互中, 还是有许多需要学习的内容。

应用状态 (Application State)

图 1-7 是一个状态图，它从浏览器的角度展示了 Alice 的完整的探索之旅。

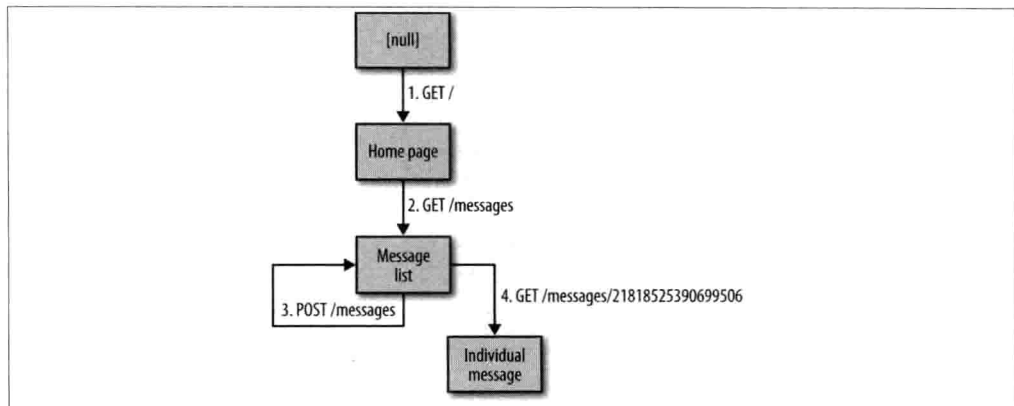


图1-7 Alic的探险: 客户端的角度

当 Alice 刚启动她手机上的浏览器的时候，浏览器没有加载任何页面，界面上还是一片空白。然后，Alice 输入了一个 URL 并且通过 GET 请求将浏览器带到了网站的主页。Alice 单击了一个链接，第二个 GET 请求将浏览器带到了消息列表页面。她提交了一个表单，这触发了第三个请求（一个 POST 请求）。对应的响应是一个 HTTP 重定向，这个重定向是由浏览器自动做出的。最后，Alice 的浏览器停留在了一个显示有 Alice 发布的消息的页面。

这张图里面的每个状态都对应到 Alice 的浏览器窗口打开的一个特定的页面（或者没有页面）。在 REST 的世界里，我们将这些信息（比如你停留在哪个页面？）称为应用状态（application state）。

当你上网的时候，你从一个应用状态转换到另一个应用状态，这个过程都对应于你单击的一个链接或者提交的一个表单。不是所有的状态之间的转换都是可用的。Alice 不能在主页直接提交一个 POST 请求，因为主页没有提供让浏览器生成 POST 请求的表单。

资源状态 (resource state)

图 1-8 是从 web 服务器的角度展示 Alice 的探索之旅的状态图。

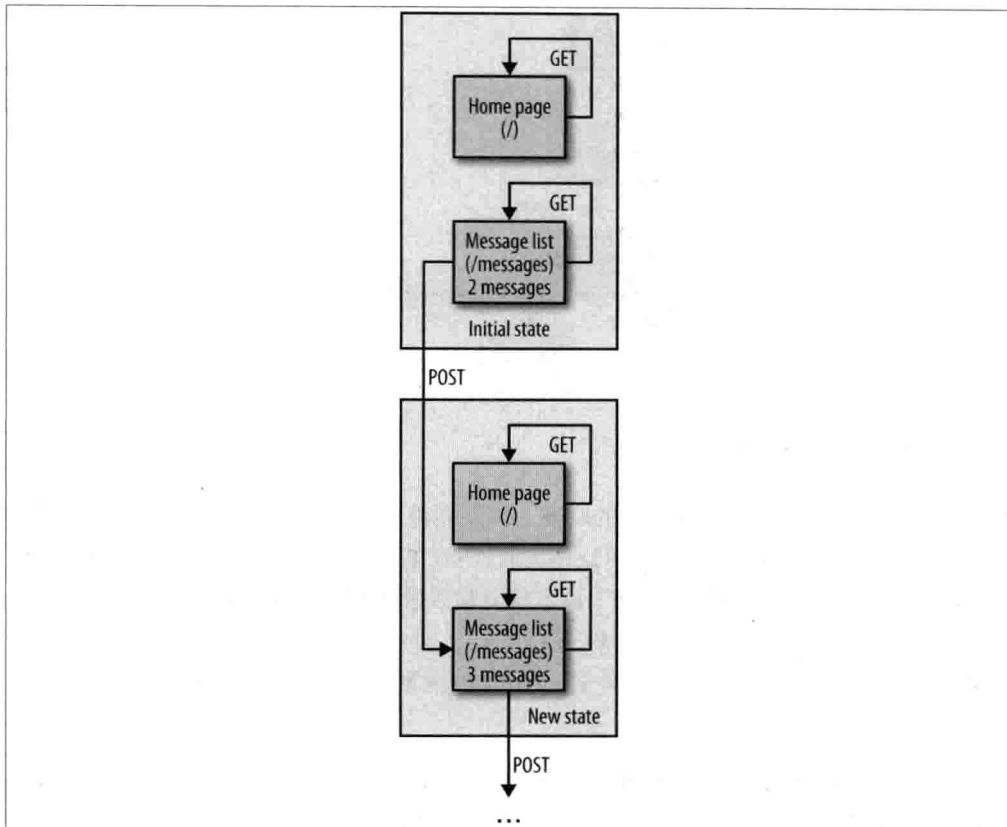


图1-8 Alice的探险：服务器的角度

服务器管理了两个资源：主页（通过“/”提供服务）和消息列表（通过“/messages”提供服务）。（服务器同时也将每条单独的消息作为资源进行管理，我为了简化状态从图中省略掉了这些资源。）简单来说，这些资源的状态就叫作资源状态。

故事开始的时候，消息列表中有两条消息：“Hello”和“Later”。浏览器向主页发送 GET 请求不会改变资源状态，因为主页是一个永远不会变化的静态文档。浏览器发送 GET 请求到消息列表页也同样不会改变资源的状态。

但是，当 Alice 向消息列表发送了一个 POST 请求的时候，这个请求将服务器切换到了一个新的状态。现在消息列表包含了三条消息：“Hello”、“Later”以及“Test”。现在已经没有办法回到原来的状态了，但是这个新的状态和原来的状态还是很相似的。和以前

一样，向主页和消息列表发送 GET 请求不会引起任何变化，但是向消息列表发送一个新的 POST 请求会使得增加第四条消息到列表里面。

由于 HTTP 会话非常短，所以服务器不知道客户端的应用状态的任何信息。客户端也不能直接控制资源状态——所有的资料都保存在服务器端。然而，网站却正常运行着。网站是通过 REST- 表述性状态移交（representational state transfer）工作的。

应用状态保存在客户端，但是服务器端可以通过发送表述（representation）——HTML 文档来操纵它。在这种情况下，提交的这个表单描述了可能的状态转换（state transition）。资源状态是保存在服务器端的，但是客户端可以通过向服务器发送表述（representation）——提交一个 HTML 表单来操作它，在这种情况下，这个提交的表单描述了客户端所期望的新的状态。

连通性（connectedness）

在这个故事中，Alice 向 YouTypeItWePostIt.com 发送了 4 个 HTTP 请求，并且获得了 3 个 HTML 文档作为返回结果。尽管 Alice 没有一一单击这些文档的所有链接，但是我们可以使用那些链接来从客户端的角度构建一幅粗略的网站地图。

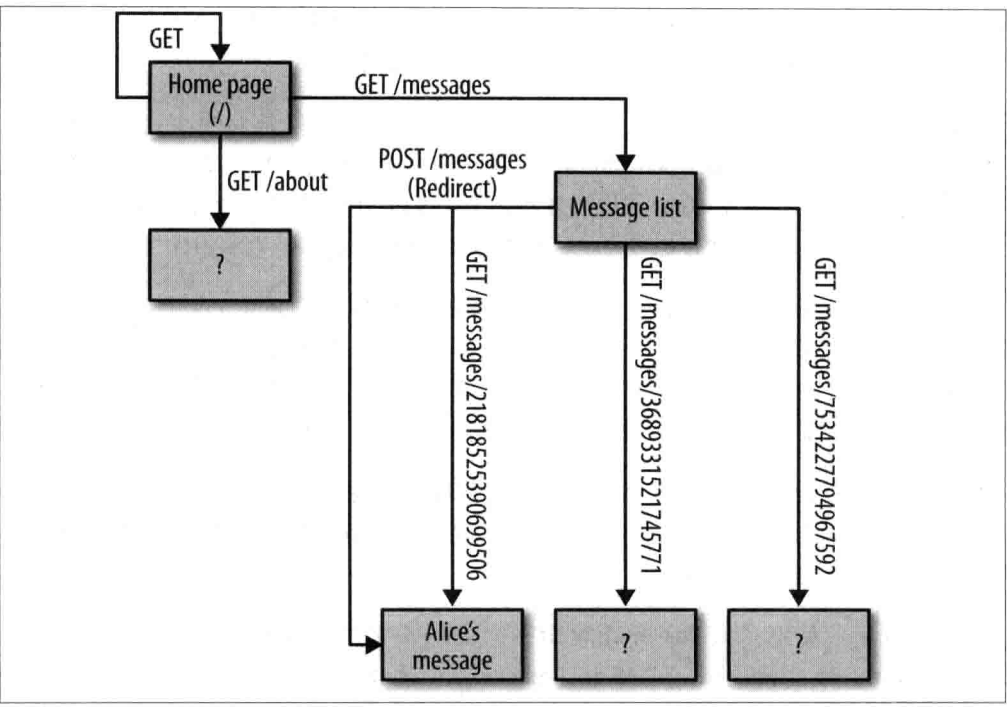


图1-9 客户端所看到的事物

这是一个由 HTML 页面组成的网。这个网间的连线是 HTML `<a>` 标签和 `<form>` 标签，它们分别描述了 Alice 可能会发起的各个 GET 或者 POST 的 HTTP 请求。我将此称为连通性原则：每个网页会告诉你如何获取相邻的网页。

14 > 网络作为一个整体按照连通性原则运转，这个原则更为人所熟知的叫法是“将超媒体作为应用状态引擎（hypermedia as the engine of application state）”，有时候简写为 HATEOAS。我更倾向于“连通性”或者“超媒体约束”，因为“将超媒体作为应用状态引擎”听起来比较吓人。但是现在，你应该没有理由害怕了。你现在已经知道什么是应用状态——简单说就是客户端当前处在哪个网页上面。超媒体是对类似于 HTML 链接、表单等的事物抽象出来的通用术语，服务器端可以通过这种技术来向客户端说明下一步的操作。

我们所说的超媒体是应用状态的引擎，其实就是说我们都是通过填写表单以及访问各种链接来浏览 Web 的。

与众不同的Web

Alice 的故事看起来并不怎么吸引人，因为万维网在过去的 20 年间已经成为最主流的互联网应用。但是在 20 世纪 90 年代，这是一个非常令人激动的故事。如果你将万维网和其他早期的竞争者相比较的话，你会发现它们的不同之处。

Gopher 协议（在 RFC 1436 中定义）看起来很像 HTTP，但是它缺少可寻址性。它没有一个简洁的方式来标识 Gopherspace 里面的一个特定文档。最起码在万维网为 Gopherspace 考虑和发布 URL 标准（首次定义在 RFC 1738）之前并不存在这种寻址的能力。这个后来发布的 URL 标准提供了和 `http://` 类似的 `gopher://` 的 URL 方案。

FTP，这个在 web 出现之前用于文件传输的非常流行的协议（RFC959 中定义）也缺乏寻址性。在 RFC1738 定义 `ftp://` 这样的 URL 方案之前，根本没有一个机器可读的方案可以指定某个 FTP 服务器上面的一个文件。你不得不长篇大论地来解释这个文件到底在哪里。为了定位服务器上的一个文件就要花费很大的精力，这是极大的浪费。

FTP 也是一个长会话的协议。一个普通的用户可以登录到 FTP 服务器上并无限期地占用服务器的一个 TCP 连接。与之对照的是，就算是一个“持久性”的 HTTP 连接最多也不会占用一个 TCP 连接超过 30 秒。

20 世纪 90 年代见证了太多的用于检索不同类型的文档和数据库的互联网协议，像 Archie、Veronica、Jughead、WAIS 和 Prospero。但是最终证明，我们不需要所有的这些协议。我们仅仅需要的是能够向不同类型的网站发送 GET 请求。所有的那些协议都渐

渐消亡了或者被网站取而代之。它们那些复杂的特定协议的规则都合并为统一的 HTTP GET。

一旦 Web 接管了这些，就更难以证明创建新的应用协议的合理性了。为什么要在你可以搭建一个每个人都可以使用的网站的时候，开发一种只有计算机专业人员才能理解的工具呢？所有成功的后 Web 协议（post-Web protocols）都是在做一些 Web 做不了的事情：P2P 协议比如 BitTorrent，实时协议比如 SSH。对于大部分的目的，HTTP 已经足够使用了。

Web 的这种空前的灵活性都来源于 REST 原则。在 20 世纪 90 年代，我们已经发现 Web 比其他的竞争者工作得更好。在 2000 年，Roy T. Fielding 的博士论文^{注 1} 解释了其中的奥秘，并将其精简为“REST”这个术语。

Web API 落后于Web

Fielding 的文章也花了很大篇幅解释 21 世纪初的许多 Web API 的问题。我前面介绍的那个简单网站比当今大部分网站部署的 Web API 或者自称的 REST API 要精致得多。如果你曾经设计过 Web API，或者为这些 API 写过客户端程序，你肯定遇到过一些下面的问题。

- Web API 经常有大量的阅读文档来告诉你如何为不同的资源构造 URL。其实这就像花很大篇幅来告诉你如何在 FTP 服务器上找到一个指定的文件一样。如果网站是这样做的，没有人会愿意使用这个网站。
和每次告诉你要往浏览器输入什么 URL 不同，Web 通常都是在 <a> 和 <form> 这样的超媒体控件中嵌入 URL，你可以通过单击链接或者提交表单来激活它。
在 REST 的世界中，将有关构造 URL 的信息放到单独的阅读文档中违背了连通性和自描述信息的原则。
- 许多网站都有帮助文档，但是你上次阅读这些文档是什么时候的事情了？除非有很严重的问题（比如你想要提交一些信息，但是所有的尝试都以失败告终），更简单的方法是随便点击网站的一些链接，并通过浏览这些服务器发送给你的这些互相连接的、能够自我描述的 HTML 文档来确定网站是如何运作的。
而如今的 API 呈现资源的方式更像是一个巨型的选项菜单，而不是一张相互连通的网。这使得很难了解资源之间的相互影响。
- 要集成一个新的 API 不可避免地需要编写新的定制化软件，或者安装别人编写的一次性的代码库。但是当你要访问一个新的网站的时候，你并不需要为此而编写任何的定制化的软件。你看到广告牌上的一个 URL，你直接将这个 URL 输入到你的浏览器就可以了，对于世界上所有的网站而言，你都可以使用同一个浏览器

注 1 Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. 博士论文，加利福尼亚大学欧文分校，2000.

来访问它们。

我们不可能做到让一个 API 客户端能理解世界上所有的 API，但是当今很多的客户端包含着许多实际上应该进行重构成为通用库的代码。这只有在 API 提供了自描述的表述的前提下才有可能实现。

- 当 Web API 发生了变化以后，定制化的 API 客户端就不能正常使用了，并且需要维护者为此进行一些代码修复。但是当网站经过重新设计改版以后，用户可能会抱怨一段时间，然后慢慢适应新的版式。他们的浏览器在此期间不会停止工作。

在 REST 的世界中，网站的改版是封装在服务器提供的自描述的 HTML 文档中的，所以，一个能理解旧版的文档的客户端也是能够理解新的版式的。

这些就是本书中试图去解决的一些问题。好消息是，在以前，这样的情况更加严重和恶劣。在以前，用不安全的方式使用安全的 HTTP 方法设计 REST API 或者将应用状态和资源状态混合使用的情况比比皆是。现在这些情况已经比较少见到了。现在的设计已经好多了，但是还可以更好。

语义挑战

现在该讲一下负面的消息了。正如文章前面的故事——Alice 浏览网站的故事，Alice 访问网站的过程非常顺利，这要归功于一个运行速度很慢但是又非常昂贵的硬件：Alice 本人。每次浏览器显示了一个网页，Alice，这个人类就会去浏览这个渲染过的页面，然后决定下一步的操作。Web 就是在人类不断地决定单击哪个链接以及填写哪个表单的过程中运行着。

Web API 的目标是在没有人类参与的前提下完成相应的工作。但是我们该如何编写程序让计算机来决定单击哪个链接呢？计算机可以解析 HTML 标记 `Get started`，但是它并不能理解“Get started”这个词组。如果提供的自描述信息不能被软件理解，我们又何苦设计这种提供自描述信息的 API 呢？

Web API 设计的最大的挑战就是：消除“理解文档的结构”和“理解文档的含义”之间的语义鸿沟（semantic gap）。简单来讲，我将其称为是：语义挑战（semantic challenge）。现在，这方面的进展非常小，我们也不可能完全解决它。好消息是到现在为止，正是由于这方面的研究进展很少和有限，所以做出一些成绩还是比较容易的。我们现在要做的是开始一起工作，而不是重复对方的工作。

因为我谈到了 Web 的技术以及如何将这些技术应用到 API 设计，所以我会后面的几章里对语义挑战方面的内容进行介绍。在第 8 章，我们会使用一些必要的工具来直接处理语义挑战的问题。

一个简单的API

在第1章中，我展示了一个非常简单的微博网站，它的网址是 <http://www.youtypeitwepostit.com/>。与此同时，我也为该网站设计了一个可编程的API，你可以亲临 <http://www.youtypeitwepostit.com/api/> 来访问它。

这个理想中的API具备了一些似曾相识的特性，而正是这些特性使得互联网简单易用。作为一名开发者，仅仅凭借在广告牌上看到的URL，你便可以理解如何使用它。

让我们将这个想象的场景继续延伸下去，同时来看看API是如何工作的。首先，假设你使用你的可编程客户端程序向广告牌上的URL发起了一次GET请求——这等效于将URL手动敲入web浏览器地址栏。至此你的客户端程序将开始接管通信，并对响应中的可用选项进行检查。它随后将可能访问响应中的链接（不一定是HTML链接）、填充表单（不一定是HTML表单）并最终完成你分配给它的任务。

本书并不会带领大家完全达成上述的理想目标。有些问题是我无法在一本书内解决的，比如：围绕因为标准缺失而产生的相关问题、工具支持方面现有水平的局限性以及计算机无法像人类般聪明的残酷事实。但是我们会一起向着这个目标进行深入地探索——其深度将远远超乎你的想象。

我已经说过，我们已经有了一个实实在在的微博API，它位于 <http://www.youtypeitwepostit.com/api/>。如果你是个具有探索精神的家伙，请尝试着编写一些代码来使用该API做些什么。看看在只知道这个URL的情况下，你能够对该API理解到什么程度。其实你在浏览各种网站时曾经完成过相同的事：你开始的时候所知道的全部信息仅仅是主页的URL，而你却可以由此逐渐完全理解这个网站。而如今只通过一个API，你又可以走多远呢？

如果你不愿意冒险又或者在为 web API 编写客户端程序方面没有太多经验的话（如果你在很久的未来才来阅读本书，我很可能已不再托管这个网站），我们将一起来完成这项工作。而第一步就是要获取该 API 首页的表述。

18 HTTP GET：安全的投注

如果你得到了一个以 `http://` 或 `https://` 开头的 URL，但是你并不知道该 URL 的另一头是什么，你首先可以做的就是向它发起一个 HTTP GET 请求。在 REST 的世界里，你除了知道指向资源的 URL 以外其他一无所知。你需要进一步去发现你的可选项，这意味着你需要从该资源处获取一个表述。这便是 HTTP GET 的作用所在。

你可以通过使用某种编程语言编写代码来发起 GET 请求，但是当我们只是对某个陌生 API 进行初步探测时，采用像 Wget 这样的命令行工具通常会更容易些。在下面的示例中，我使用了 `-S` 和 `-O` 两个参数选项，前者将打印出来自服务器的完整的 HTTP 响应内容，而后者将以打印出响应文档内容的方式替换原来将响应内容保存为文件的方式。^{注 1}

```
$ wget -S -O - http://www.youtypeitwepostit.com/api/
```

上面的命令将向服务器发起一个如下的请求：

```
GET /api/ HTTP/1.1
Host: www.youtypeitwepostit.com
```

HTTP 规范告诉我们，GET 请求是为了获取一个表述而作的一次请求。该种类型的请求主观上并没有去改变服务器上资源状态的意图。这就是说如果你得到了一个 URL 而没有更多信息可供参考时，你总是可以向它发起一个 GET 请求从而得到一个资源表述作为回报。你的 GET 请求将不会造成如删除所有数据这样的破坏性效果，因此我们说 GET 是一个安全的方法。

当然我们也允许服务器在处理 GET 请求时改变一些附属性的事物，例如递增计数器或将请求日志记录到文件中，但是这并非是 GET 请求原本的用途。没有人会仅仅为了递增计数器而去使用 HTTP 请求。

而在现实生活中，我们并不能保证 HTTP GET 请求是安全的。一些较早的设计会强制你使用 HTTP GET 请求来删除数据，但是这种糟糕的特性在新近的设计中相当少见。大部分的 API 设计者现在都能理解：客户端之所以频繁向 URL 发起 GET 请求只是为了看看

注 1 `-O` 参数在 wget 帮助文档中的描述是将响应内容写入指定的文件，这与作者的描述看似不符。但是当没有指定输出文件名时，wget 会将内容输出到默认输出流，即命令行控制台。若不指定 `-O`，wget 会根据请求的资源名称生成默认的文件，并将响应内容写入该文件，所以这与作者的描述的效果是一致的。——译者注

该 URL 背后的内容是什么。在设计时给 GET 请求赋予重大的副作用是不合理的。

如何读取HTTP响应

为了响应我发起的 GET 请求，服务器发送了如下所示的数据块：

```
HTTP/1.1 200 OK
ETag: "f60e0978bc9c458989815b18ddad6d75"
Last-Modified: Thu, 10 Jan 2013 01:45:22 GMT
Content-Type: application/vnd.collection+json

{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/",
    "items" : [
      { "href" : "http://www.youtypeitwepostit.com/api/
messages/21818525390699506",
        "data": [
          { "name": "text", "value": "Test." },
          { "name": "date_posted", "value": "2013-04-22T05:33:58.930Z" }
        ],
        "links": []
      },

      { "href" : "http://www.youtypeitwepostit.com/api/messages/3689331521745771",
        "data": [
          { "name": "text", "value": "Hello." },
          { "name": "date_posted", "value": "2013-04-20T12:55:59.685Z" }
        ],
        "links": []
      },

      { "href" : "http://www.youtypeitwepostit.com/api/messages/7534227794967592",
        "data": [
          { "name": "text", "value": "Pizza?" },
          { "name": "date_posted", "value": "2013-04-18T03:22:27.485Z" }
        ],
        "links": []
      }
    ],
    "template": {
      "data": [
        { "prompt": "Text of message", "name": "text", "value": "" }
      ]
    }
  }
}
```

通过以上内容我们能获悉到哪些信息呢？首先，每一个 HTTP 响应可以被分成 3 个部分：状态码，有时我们也称它为响应码

这部分是由三位数字组成的，它简要说明了请求目前的进展。响应码是 API 客户端从响应中最先看到的信息，它奠定了响应剩余部分的基调。以上的示例中，我们看到的状态码是 200 (OK)，这是客户端所期盼的——这意味着一切进展顺利。

在附录 A 中，我对所有的 HTTP 响应码进行了说明，同时还进行了一些有价值的扩展。

实体消息体 (entity-body)，有时我们也称它为消息体 (body)

这部分是一个采用某种数据格式书写成的文档，并且我们预期该文档是可以被客户端所理解的。如果你将 GET 请求理解成为获取表述而发起的一次请求，那么你可以将实体消息体理解为你最终得到的表述（严格来说，整个 HTTP 响应都是“表述”，但是重要的信息通常都记录在实体消息体中）。

20

在本例中，实体消息体是响应中最后较大那部分文档，其中充斥着很多花括号。

响应报头

响应报头的发送顺序排在状态码和实体消息体之间，通常是一系列用于描述实体消息体和 HTTP 响应的键值对。在附录 B 中，我将对所有的标准 HTTP 报头进行说明，并做了一些有助于理解的延伸。

最重要的 HTTP 报头是 Content-Type，它向 HTTP 客户端说明了如何去理解实体消息体。因为它非常重要，所以它的值都具有特定的名称。我们将 Content-Type 报头的值称为实体消息体的媒体类型 (media type)（它同样也可以被称为 MIME 类型或 *content type*，有时“media type”是可以带有连接符的：*media-type*）。

就平时人们通过浏览器就能看到的 Web 而言，最常见的媒体类型是 `text/html`（针对 HTML）以及图片类型，例如 `image/jpeg`。而在此例中的媒体类型，你或许闻所未闻：`application/vnd.collection+json`。

JSON

如果你是一个 web 开发人员，你或许已经认出该实体消息体其实是一个 JSON 文档。如果你没有，以下是一段为你准备的关于 JSON 的快速介绍。

JSON，参见 RFC 4627 中的表述，它是一种使用普通文本来表示简单数据结构的标准。它使用双引号来描述字符串：

"this is a string"

它使用方括号来描述列表：

[1, 2, 3]

它使用花括号来描述对象（键值对的集合）：`{ "key" : "value" }`

JSON 数据看上去非常像 JavaScript 或 Python 的代码。JSON 标准将约束建立在普通文本之上，它认为一个像 *It was the best of times* 这样的光秃秃的字符串是不能被接受的，即便任何人都能看到并理解它。要想成为合法的 JSON 数据，字符串必须用双引号括起来：*"It was the best of times."*

Collection+JSON

◀ 21

这么说来，该实体消息体文档就是 JSON 格式了，对吗？请别着急！你可以将这个文档交由 JSON 解析器来顺利地进行解析，但是这并不是 web 服务器希望你所做的。以下是服务器对你说的：

Content-Type: application/vnd.collection+json

这便与 JSON RFC 所描述的不符了，JSON 文档应该以 `application/json` 的类型提供，像这样：

Content-Type: application/json

那么 `application/vnd.collection+json` 又是何方神圣？很明显，这种格式也是基于 JSON 的，因为它看起来很像 JSON 并且它的媒体类型名中也含有“json”的字样。那它到底是什么呢？

如果你尝试在 web 中搜索“`application/vnd.collection+json`”，你将会发现它是一种注册为 Collection+JSON^{注2}的媒体类型。当你向 `http://www.youtypeitwepostit.com/api/` 发起一个 GET 请求时，你不会得到任何的 JSON 文档——你得到的其实是一个 Collection+JSON 文档。

在第 6 章中，我将会详细地讨论 Collection+JSON，而眼下先做个简短的介绍。Collection+JSON 是一个用于在 Web 上发布资源的可搜索列表的标准。JSON 将约束建立在普通文本之上，而 Collection+JSON 则将约束建立在 JSON 之上。服务器不仅能提供像 `application/vnd.collection+json` 这样的任意 JSON 文档，它也可以只提供 JSON 对象：

注 2 Collection+JSON 是一种在该页中定义的个人标准 (<http://amundsen.com/media-types/collection>)。

```
{}
```

但是又不仅仅是一个对象，这个对象还必须拥有一个称为 `collection` 的属性，该属性可以映射到另一个对象：

```
{"collection": {}}
```

而该 “`collection`” 对象应该具有一个称为 `items` 的属性，该属性映射到了一个列表：

```
{"collection": {"items": []}}
```

在 “`items`” 列表属性中的项目也必须是对象：

```
{"collection": {"items": [{}, {}, {}]}}
```

如此反复，约束相承。最终你将会得到一个如你所见的高度格式化的文档。它的开始部分如下：

```
{ "collection":  
  {  
    "version" : "1.0",  
    "href" : "http://www.youtypeitwepostit.com/api/",  
    "items" : [  
  
      { "href" : "http://www.youtypeitwepostit.com/api/messages/21818525390699506",  
        "data": [  
          { "name": "text", "value": "Test." },  
          { "name": "date_posted", "value": "2013-04-22T05:33:58.930Z" }  
        ],  
        "links": []  
      },  
  
      ...  
    ]  
  }  
}
```

通过总览这个文档，所有这些约束的用途变得逐渐清晰起来。`Collection+JSON` 是提供列表的一种方式，我指的列表并非标准 `JSON` 也可以提供的那种列表数据结构，而是一种描述 `HTTP` 资源的列表。

`collection` 对象拥有一个 `href` 属性，而它的值是一个 `JSON` 字符串。但是它不仅仅是一个字符串——它是我向其发起 `GET` 请求的 `URL` 地址：

```
{ "collection":  
  {  
    "href" : "http://www.youtypeitwepostit.com/api/"  
  }  
}
```

`Collection+JSON` 标准将该字符串定义为“用于获取一个文档表述的地址”（换句话说，

它是 collection 资源的 URL)。collection 的 items 列表中的每个对象都有其自己的 href 属性，并且每个值都是一个包含了一个 URL 的字符串，比如 `http://www.youtypeitwepostit.com/api/messages/21818525390699506`（换句话说，列表中的每一项都代表了一个拥有 URL 的 HTTP 资源）。

一个没有遵守以上规则的文档将不能算是一个 Collection+JSON 文档：它只能算是某种 JSON。通过遵守 Collection+JSON 的约束，你将获得使用资源和 URL 这些概念的能力。在 JSON 中只能使用像字符串和列表这样的简单元素，而以上这些概念在 JSON 中是没有定义的。

向API写入数据

我该如何使用 API 来向微博发布一条消息呢？以下是 Collection+JSON 规范里所描述的：

如果想要在 collection 中创建一个新的 item 项，客户端首先要使用**模板**对象来组装一个有效的 item 表述，然后使用 HTTP POST 来向服务器发送该表述以获得处理。

以上所述并不能算是一个按部就班的描述，但是它指明了答案的方向。Collection+JSON 的工作方式与 HTML 相似，在 HTML 中，服务器向你提供了各种表单（就好比 Collection+JSON 中的模板），你可以填充这些表单来创建文档，然后使用 POST 请求来向服务器发送该文档。

23

再次说明，第 6 章将会具体讨论 Collection+JSON，而随后我只会快速地介绍下相关的内容。我们再来查看下我早前向大家展示的那个对象。它的 `template` 属性便是在 Collection+JSON 规范中提及的“模板对象”。

```
{
  ...
  "template": {
    "data": [
      {"prompt": "Text of message", "name": "text", "value": ""}
    ]
  }
}
```

我们来填写这个模板，我将 value 对应值中的空白字符串替换成了我想要发布的内容：

```
{ "template":
  {
    "data": [
      {"prompt": "Text of the message", "name": "text", "value": "Squid!"}
    ]
  }
}
```

随后我将这个填充完毕的模板作为 HTTP POST 请求的一部分进行发送：

```
POST /api/ HTTP/1.1
Host: www.youtypeitwepostit.com
Content-Type: application/vnd.collection+json

{ "template":
  {
    "data": [
      { "prompt": "Text of the message", "name": "text", "value": "Squid!" }
    ]
  }
}
```

(请注意，我请求中的 Content-Type 是 application/vnd.collection+json。该模板在填充后仍是一个有效的 Collection+JSON 文档。)

服务器做出了应答：

```
HTTP/1.1 201 Created
Location: http://www.youtypeitwepostit.com/api/47210977342911065
```

此处的 201 响应码 (Created) 相比 200 (OK) 稍显特殊；它表示一切进展顺利并且在本次响应中已针对我当次请求创建了一个新的资源，而 Location 报头给出了新生资源的 URL。

24 在第 1 章中，Alice 曾使用 web 界面在微博网站上发过帖子。而我现在已经成功使用该网站的 web API 完成了相同的事情。

HTTP POST:资源是如何生成的

为了向 collection 添加一个新的 item，你向 collection 的 URL 发送了一个 POST 请求。这里所涉及到的不仅仅是 Collection+JSON 是如何工作的内容，这里还涉及到了 HTTP 中有关 POST 的一个基本事实。RFC 2616，即 HTTP 规范中是这样描述 POST 的：

我们将 POST 设计为一个具备如下功能的统一方法：

- 对现有资源的注解。
- 向布告栏、新闻组、邮件列表或类似的文章组发布消息。
- 向数据处理流程提供例如表单提交结果的数据块。
- 通过追加操作来扩充数据库。

其中提到的第二个重点功能，“向布告栏、新闻组、邮件列表或类似的文章组发表消息”便准确地覆盖到了微博。

我所发出的 POST 请求看上去非常像一个 HTTP 响应，因为它具有 Content-Type 报头和实体消息体。虽然我在早前所展示的 GET 请求中没有提供任何 HTTP 报头，但事实上任何 HTTP 请求都可以具有报头，甚至有好几个报头（比如 Accept）对于 GET 请求来说是非常重要的。后续我将会在这些特别重要的 HTTP 报头出现的时候进行重点讨论，但是请务必去附录 B 查阅标准 HTTP 报头的完整列表。

让我们继续往下走。再次回顾，以下是我通过 POST 请求获取的响应：

```
201 Created
Location: http://www.youtypeitwepostit.com/api/47210977342911065
```

当你收到一个 201 响应码时，后面的 Location 报头将会告诉你应该去何处找寻你刚才所创建的内容。RFC2616 详细说明了 201 响应码和 Location 报头的含义，但是为了清晰起见，Collection+JSON 规范同样也提及了这些内容。

如果我发送了如下的 GET 请求：

```
GET /api/47210977342911065 HTTP/1.1
Host: www.youtypeitwepostit.com
```

我将会看到熟悉的面孔：

```
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/47210977342911065",
    "items" : [

      { "href" : "http://www.youtypeitwepostit.com/api/messages/47210977342911065",
        "data": [
          { "name": "date_posted", "value": "2014-04-20T20:15:32.858Z" },
          { "name": "text", "value": "Squid!" }
        ],
        "links": []
      }
    ]
  }
}
```

25

我们通过一个完整的 application/vnd.collection +json 文档来表示这个单独的微博帖子。它是一个 collection，但是它的 items 列表只含有一个列表项。而这个经过填充的模板同样也是一个有效的 application/vnd.collection +json 文档，即使它根本没有使用 collection 属性。

这是 Collection+JSON 的一个便利的特性。几乎文档中的所有内容都是可选的。这意味着你无须为了处理不同类型的文档而编写不同的解析器。Collection+JSON 使用相同的 JSON 格式来表示列表项、单独项、已填充的模板及搜索结果。

由约束带来解放

在 RESTful 设计中，一个违反直觉的经验就是：约束成为了一种解放手段。HTTP GET 方法的安全约束就是一个很好的例子。因为有了这个安全约束，当你还不知道可以对某个 URL 做些什么时，你总是可以先向它发送 GET 请求从而看看它的表述。就算最后发现这样做没有带来什么帮助，也不会产生任何糟糕的副作用，因为你所使用的是一个 GET 请求。这便是一个对解放的承诺，而唯一让这成为可能的原因是我们在服务器端制定了更加苛刻的约束。

如果服务器向你发送了一个普通文本文档，该文档中记录着 9，你将无法知晓它表示的是一个数字 9 还是一个字符串“9”。但是如果你得到的是一个 JSON 文档，那么你可以确信文档中所记录的 9 是一个数字。JSON 标准对文档的含义进行了约束，这样便为客户端和服务端进行有意义的交流提供了可能性。

在过去的若干年里，数以百计的公司都经历过以下的思考：

1. 我们需要一个 API。
2. 我们将使用 JSON 作为文档的格式。
3. 我们将使用 JSON 来发布我们的事物列表。

26

以上的三点想法都不错，但是不足以说明我们的 API 应该是什么样的。最终的结果就是数以百计的 API 表面上很相似（因为它们都是使用 JSON 来发布事物列表的！）但却无法完全兼容。对一个 API 的学习经验无法帮助客户端去理解下一个 API。

这意味着我们需要更多的约束，而 Collection+JSON 标准为我们提供了更多的约束。如果我选择使用自定义的 API 设计而非 Collection+JSON 的话，那么在我列表中的一个独立的项将可能看上去是以下这样的：

```
{
  "self_link": "http://www.youtypeitwepostit.com/api/messages/47210977342911065",
  "date": "2014-04-20T20:15:32.858Z",
  "text": "Squid!"
}
```

反之，如果我遵守了 Collection+JSON 的约束，那么一个独立的项看上去是这样的：

```
{ "href" : "http://www.youtypeitwepostit.com/api/messages/1xe5",
```

```

    "data": [
      { "name": "date_posted", "value": "2014-04-20T20:15:32.858Z" },
      { "name": "text", "value": "Squid!" }
    ],
    "links": []
  }
}

```

自定义设计看上去确实更加简洁，但这并不非常重要——JSON 的压缩率原本就已经很好了。在换成 Collection+JSON 这种紧凑性稍逊的表述后，我却得到了大量更有用的特性：

- 我无须再向所有的用户说明 href 的值是一个 URL，而且我也无须对什么是 URL 进行说明。Collection+JSON 标准已经说明了一个 item 项的 href 包含了指向该 item 项的 URL。
- 我无须为了向用户说明 text 表示的是消息的文本内容而特地编写一个供人们阅读的独立文档。这些信息会直接展现在实际需要的地方——即出现在你在投递消息时所要填写的模板中。

```

    "template": {
      "data": [
        { "prompt": "Text of the message", "name": "text", "value": null }
      ]
    }
  }
}

```

- 任何理解 application/vnd.collection+json 的代码库都可以自然而然地懂得如何使用我的 API。如果我选择了一种自定义的设计，那我将只能基于仅有的 JSON 解析器和 HTTP 代码库来编写全新的客户端代码，或者让我的用户们自行编写这些代码。

◀ 27

通过采纳 Collection+JSON 的约束，我从原本需要编写的一大堆的文档和代码中解放出来，同时我也将我的用户从学习一个接一个的自定义 API 中解放了出来。

应用语义所产生的语义鸿沟

当然，Collection+JSON 约束并不能约束所有的事情。Collection+JSON 并没有指定 collection 中的 items 应该是具有 date_posted 字段和 text 字段的微博帖子。这部分的工作是我完成的，因为我想为本书设计一个简单的微博实例。如果我选择以“烹饪书”来作为实例的话，我仍然可以使用 Collection+JSON，但是 items 的字段将可能换成是 ingredients（配料）和 preparation_time（准备时间）了。

我将这些设计中的额外信息称为应用语义（application semantic），因为它们在应用之间有着很大的差别。应用语义是引起我在第 1 章中所提及的语义鸿沟的原因。

如果要设计一个真正的微博 API，我会选择远比 `text` 和 `date_posted` 更加复杂的应用语义。就该 API 自身来说，这没什么问题。但是时下有很多公司都在设计着各种微博 API，因此也冒出了很多的设计，可这些设计的应用语义是相互不兼容的，由此产生了很多不同的语义鸿沟。所有这些公司都在以不同的方式做着相同的事情，它们的用户需要编写不同的软件客户端来完成相同的任务。

了解到 Collection+JSON 无法处理语义兼容问题这一现实并不意味着我们不再需要使用 Collection+JSON，兼容性只是一个度的问题。从我们在 20 世纪 90 年代停止发明自定义的 Internet 协议并对 HTTP 进行标准化开始，我们已经向兼容性迈进了一大步。如果我们全都赞同以 JSON 文档提供服务，虽然从技术上来说这未必是个好主意，但是这样确实能缩窄我们的语义鸿沟，而标准化地采用 Collection+JSON 将会更进一步地改善这个问题。

如果微博 API 的发布者可以走到一起并协议使用一组通用的应用语义，那么微博的语义鸿沟将荡然无存（这将可能涉及到 *profile*，我会在第 8 章中讨论这块内容）。一旦我们共享更多的约束，我们将可以设计出更加兼容的接口，语义鸿沟也会变得更小，而我们的用户将受益更多。

28 也许你并不希望自家的 API 与对手的 API 进行互相协作，从而人为地扩大了语义鸿沟。但是在 API 差异化方面我们有比这种办法更好的方式。我在本书中的目标便是让你能专注于 API 中语义鸿沟藏身的部分，为填平鸿沟提供一些新的方式，因为从未有人尝试过这样做。

资源和表述

到现在，我已经展示了两个 REST 实例：一个是网站（见第 1 章），一个是 web API（见第 2 章）。我都是以案例的形式来展开论述的，因为对于 REST 而言，并不存在一套 RFC 标准，在这一点上，它与 HTTP 或 JSON 有所不同。

REST 不是一种协议，也不是一种文件格式，更不是一种开发框架。它是一系列的设计约束的集合：无状态性、将超媒体作为应用状态的引擎等，我们将这些约束统称为 *Fielding* 约束，因为它们最早是由 Roy T. Fielding 博士在 2000 年的时候发表的关于软件架构的论文中提出的，这篇论文将这些内容整理到一起命名为“REST”。

“REST”这个术语的受欢迎程度现在已经远远超出了其在 Fielding 的论文中的重要性。在 Fielding 的论文中，REST 主要是作为一个案例来将那些万维网中习以为常的事物与一个更具有广泛意义的设计过程连接到一起。之所以 REST 会流行，是由于这个术语描述了人类历史上最为成功的技术之一：万维网的架构。

在本章中，我会从万维网的方面来讲解 Fielding 约束。我所引用的“圣经”并不是 Fielding 的论文（你可以在附录 C 中了解更多内容，Fielding 关于以 API 为中心的讨论），而是 W3C 颁布的 Web 指南：《万维网的架构（第一卷）》（*The Architecture of the World Wide Web, Volume One*）（实际上并不存在第二卷）。Fielding 的论文解释了 Web 设计背后的决策，而《架构》一书进一步说明了发源于这些决策的三项技术：URL、HTTP 和 HTML。

我确信你知道这三项技术，但是更深层次地理解这些技术是理解 Fielding 约束的关键、是理解这些约束如何推动 Web 成功的关键，更是理解如何将这些约束应用于自己的 API 的钥匙。

潜藏在这三项 web 技术背后的两大核心理念是：资源（resource）和表述（representation）。我前面提到过它们，现在我们该仔细看看它们的庐山真面目了。

万物皆可为资源

就其本质而言，任何足够重要并被引用的事物都可以是资源。如果你的用户“想要建立指向它的超文本链接，提出或者反对关于它的断言，获取或者缓存它的表述，供另外的表述引用它的全部或者部分，给它增加注释信息，或者对它执行某些操作”（源自《万维网的架构》），你都应该将它定义为资源。

资源一般是可以保存到计算机里面的事物。比如电子文档，数据库的一条记录，或者一个算法的运行结果。《架构》一书将它们统称为“信息资源（information resource）”，因为它们的本质形式都是一串比特数据流。但是资源实际上可以是任何事物：比如一个石榴、一个人、黑色、勇气、母女关系、质数的集合。唯一的条件是每个资源必须拥有 URL。

你还记得那个东西吗？就是前些日子你拿着的那个，后来给……你知道我在说什么吗？你当然听不懂。因为我说的不够具体。我可能说的是任何一件东西。Web 也是一样的。客户端和服务端只有在对事物的命名上达成一致以后才能针对这个事物相互通信。在 Web 上，我们使用 URL 来为每个资源提供一个全球唯一的地址。将一个事物赋以 URL，它就会成为一个资源。

从客户端的角度看，它并不关心资源是什么，因为它从来看不到资源，它看到的永远只是 URL 和表述。

表述描述资源状态

一个石榴可以是一个 HTTP 资源，但是你不可能通过互联网将石榴进行传输。数据库的一条记录可以是一个 HTTP 资源；事实上，它是一个信息资源，因为你可以通过互联网将它一个字符一个字符地发送出去。但是客户端如何处理这堆来自来源不明的数据库且没有上下文信息的二进制数据呢？

当客户端对一个资源发起一个 GET 请求的时候，服务器会以一种有效的方式提供一个采集了资源信息的文档作为回应。这就是表述——一种以机器可读的方式对资源当前状态的说明。石榴的大小和成熟度、数据库字段中的数据都属于这样的说明。

对于数据库中的一条记录，服务器可以用 XML 文档、JSON 对象、逗号分隔的数值或者用来生成它的 SQL INSERT 语句来描述它。它们都是合法的表述；这依赖于客户端的请求。

有的应用程序可能使用自定义的 XML 词汇表来表示一个作为商品出售的石榴。有的程序可能会用通过摄影机拍摄的石榴照片来表示一个石榴。这都依赖于具体的程序的设计。表述可以是任何机器可读的包含资源相关信息的文档。

往来穿梭的表述

在第 2 章中，我展示了一个使用 POST 请求来发布微博的客户端。随后，这个客户端发送了一个 HTTP GET 请求来获取这个微博的表述：

```
GET /api/5266722824890167 HTTP/1.1
Host: www.youtypeitwepostit.com
```

服务器返回一个 application/vnd.collection+json 格式的表述，如下：

```
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...
{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://localhost:1337/api/",

    "items" :
    [{
      "href": "http://localhost:1337/api/5266722824890167",
      "data": [
        {
          "name": "text",
          "value": "tasting"
        },
        {
          "name": "date_posted",
          "value": "2013-01-09T15:58:22.674Z"
        }
      ]
    }
  ],

  "template" : {
    "data" : [
      {
        "prompt" : "Text of message",
        "name" : "text",
        "value" : ""
      }
    ]
  }
}
```

```
}  
}
```

32

但是这条微博还有另一种形式的表述：也就是客户端起初通过 POST 请求发送给服务器端的表述。它也是一个 `application/vnd.collection+json` 文档，内容如下：

```
{ "template":  
  {  
    "data": [  
      { "prompt": "Text of the message", "name": "text", "value": "Squid." }  
    ]  
  }  
}
```

这两个表述看起来明显不同。其中一个是在 `template` 对象中包含那些微博的基本信息，另一个是在一个 `items` 列表中包含那些信息。但是事实很清楚，它们是同一资源的两种不同的表述，它们都是那条写有“Squid”的微博的表述。

当客户端为了创建一个新的资源而发起一个 POST 请求的时候，它会发送一个表述：客户端所期望的新资源的样式。服务器端的工作就是创建这个资源或者拒绝创建这个资源。客户端的表述只是一个建议。服务器可以增加、修改或者忽略表述的任何一部分（此处，服务器对表述数据增加了一个 `date_posted` 字段）。

Web 也以同样的方式工作着。回到第 1 章，故事中的虚拟主人公 Alice 通过发送一个 POST 请求在微博网站上发布了一条微博，这个 POST 请求包含了一个 `application/x-www-form-urlencoded` 格式的表述：

```
message=Test&submit=Post
```

这条表述看起来一点也不像 Alice 之前收到的复杂的 HTML 文档，但是它们都是那条写有“Test”微博的表述。

我们通常都认为表述是服务器发送给客户端的东西，这是由于在我们上网的时候，发送的大部分的请求都是 GET 请求，我们一直都在请求获取表述。但是实际上，在 POST、PUT 或者 PATCH 请求中，客户端也会向服务器端发送表述，服务器随后的工作就是改变资源状态，这种情况下的表述反映的是将来的表述。

服务器发送的表述用于描述资源当前的状态。客户端发送的表述用于描述客户端希望资源拥有的状态。这就是表述性状态移交。

资源有多重表述

一个资源可以有很多种表述。政府的官方文档也经常有好多个语言版本。有的资源既有整

体概括性的表述，也有面面俱全的细致化的表述。有一些 API 可以使用 JSON 和 XML 数据格式来表示同一数据。当这种情况发生的时候，客户端应该如何指定它想要的表述呢？

这里有两种策略，我将会在第 11 章对它们进行更详细的描述。第一种就是内容协商 (content negotiation)，客户端通过一个 HTTP 报头的值来区分这些表述。第二种就是为资源分配多个 URL——一个 URL 对应一种表述。

就像一个人在不同的场景下会有不同的称呼一样^{注 1}，一个资源也可以由多个 URL 进行标识。当这种情况发生的时候，公开这些资源的服务器应该将其中的一个 URL 指定为“标准”URL。我将在第 11 章中介绍其中的更多细节内容。

HTTP 协议语义 (Protocol Semantics)

尽管任何事物都可以成为一个资源，但是客户端并不能随心所欲地对资源进行任意的操作。所能进行的操作是有规定的。在一个 RESTful 系统里，客户端和服务端只能通过相互发送遵循预定义协议的消息来进行交互。

在 web API 的世界里，这个协议就是 HTTP (第 13 章中的 RESTful API 架构并未采用 HTTP。) API 客户端可以通过发送一些不同类型的 HTTP 消息与 API 进行交互。

HTTP 标准定义了 8 种不同类型的消息，下面 4 个是最常用的：

GET

获取资源的某个表述。

DELETE

销毁一个资源。

POST

基于给定的表述信息，在当前资源的下一级创建新的资源。

PUT

用给定的表述信息替换资源的当前状态。

下面两个方法是客户端在分析研究 API 的时候经常使用到的：

HEAD

注 1 “你好，Mike!”，“@mamund”（网名），“晚上好，Amundsen 先生。”

获取服务器发送过来的报头信息（不是资源的表述），这些报头信息是在服务器发送资源的表述的时候被一起发送过来的。

OPTIONS

获取这个资源所能响应的 HTTP 方法列表。

另外两个定义在 HTTP 标准中的方法：CONNECT 和 TRACE 只用于 HTTP 代理，所以在本书中我并不对它们进行介绍。

34 我建议 API 设计者还要考虑第 9 个 HTTP 方法，这个方法并没有被写进 HTTP 标准中，而是作为补充内容在 RFC 5789 中被定义的：

PATCH

根据提供的表述信息修改资源的部分状态。如果有某些资源状态在提供的表述中没有被提到，这些状态就保持不变。PATCH 类似于 PUT，但是允许对资源状态进行一些细粒度的改动。

我同时还希望你了解两个正处于标准化进程中的 HTTP 扩展方法。它们是在互联网草案“snell-link-method”中定义的，我会在第 11 章进行介绍以进一步说明它们的意义：

LINK

将其他资源连接到当前资源。

UNLINK

销毁当前资源和其他某些资源的连接关系。

总体来说，就是前面介绍的这些方法确定了 HTTP 的协议语义。仅仅通过查看 HTTP 请求中所采用的方法，你就可以大概了解客户端要做什么了：它是打算获取一个资源表述？还是删除资源？又或者是将两个资源连接到一起？

但是你还做不到完全明白它们要做什么，因为资源可以是任何事物。向一个“博客日志”发送的 GET 请求和向一个“股票代码”发送的 GET 请求是类似的。这两个请求拥有相同的协议语义，但是却有完全不同的应用语义（application semantic）。HTTP 是 HTTP，但是日志 API 不会是一个股票报价 API。

我们无法仅仅通过恰当地使用 HTTP 来满足这样的语义要求，因为 HTTP 协议并没有定义任何应用语义。但是你的应用语义必须和 HTTP 的协议语义保持一致。“获取一篇日志”和“获取一个股票报价”都应该归为“获取一个资源的表述”，所以这两个请求都应该

使用 HTTP GET。

下面的章节会针对一些使用范围比较广泛的 HTTP 方法，更加详细地介绍它们在协议语义方面的内容。

GET

你肯定已经很熟悉这个方法了。客户端会通过发送 GET 请求来获取某个 URL 所标识的资源的表述。在前面的内容中，客户端请求一条微博的表述，服务器以 `application/vnd.collection+json` 格式返回了这条微博：

35

```
GET /api/45ty HTTP/1.1
Host: www.youtypeitwepostit.com

HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...

{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://localhost:1337/api/",

    "items" :
    [{
      "href": "http://localhost:1337/api/2csl73jr6j5",
      "data": [
        {
          "name": "text",
          "value": "Bird"
        },
        {
          "name": "date_posted",
          "value": "2013-01-24T18:40:42.190Z"
        }
      ]
    }
  ],

  "template" : {
    "data" : [
      { "prompt" : "Text of message", "name" : "text", "value" : "" }
    ]
  }
}
```

我前面提到过，GET 是被定义为安全的 HTTP 方法。它仅仅是对信息的一次请求。向服务器发送一条 GET 请求对资源的影响应该和没有发送 GET 请求一样——也就是，没有任何影响。像一些日志记录、速率限制等副作用是可以接受的，但是客户端绝对不要希望发起的 GET 请求能改变资源的状态。

GET 请求中最常见的响应码是 200 (OK)。此外像 300 (Moved Permanently) 这样的重定向码也是很常见的。

DELETE

当客户端想要一个资源消失的时候，它可以发送一个 DELETE 请求。客户端这时候希望服务器将资源进行销毁，并且以后再也不会提到它。当然，服务器没有义务来删除一些它不希望删除的资源。

在下面这个 HTTP 片段中，客户端要求删除一条微博：

36

```
DELETE /api/45ty HTTP/1.1
Host: www.youtypeitwepostit.com
```

服务器返回的状态码是 204 (No Content)，这个状态码表示微博已经被成功删除了，现在已经没有任何关于它的信息可讨论了：

```
HTTP/1.1 204 No Content
```

如果一个 DELETE 请求发送成功了，收到的状态码可能是 204 (No Content, 也就是，“删除成功，我没有其他关于这个资源的信息描述了”)，也可能是 200 (OK, 也就是“删除成功，这里是关于它的一条消息”) 或者 202 (Accepted, 也就是“我稍后将删除这个资源”)。

如果客户端试图 GET 一个已经被 DELETE 的资源，服务器会返回错误响应码，通常是 404 (Not Found) 或者 410 (Gone)：

```
GET /api/45ty HTTP/1.1
Host: www.youtypeitwepostit.com
```

```
HTTP/1.1 404 Not Found
```

幂等性 (Idempotence)

DELETE 很明显不是一个安全的方法。发送一个 DELETE 请求的效果不同于未发送 DELETE 请求。但是 DELETE 方法有另外一个很有用的属性：它是幂等的 (idempotent)。

一旦你删除了一个资源，这个资源就消失了。资源状态也就永久性地改变了。你可以再次发送一条 DELETE 请求，这时，你可能会收到一个 404 错误，但是资源状态和你第一

次发送 DELETE 请求之后的状态是一致的。资源还是不存在的。这就是幂等性。发送两次请求对资源状态的影响和发送一次请求的影响是一样的。

幂等性是一个很有用的特性，因为互联网不是一个可靠的网络。假设你发送了一个 DELETE 请求，然后你的连接超时了。由于你并没有收到响应信息，所以你不知道前面的 DELETE 请求是否顺利完成。你只要再次发送一条 DELETE 请求并不断重试直到收到响应信息就可以了。执行两次 DELETE 请求并不比只执行一次造成更多的影响。

幂等的概念来自于数学。零乘运算是一个幂等运算。 5×0 是 0， $5 \times 0 \times 0$ 还是 0。一旦你将一个数值乘以零，你就可以无限次地乘以零，你得到的都是同样一个结果：零。HTTP DELETE 方法就相当于用零乘以一个资源。

乘以 1 是一个安全的运算，HTTP GET 也是一个安全的方法。你可以将一个数值不停地乘以 1，数值不会产生任何变化。每个安全的运算都是幂等的。

POST-to-Append

POST 是另外一个你以前应该用到过的 HTTP 方法。我会分开介绍 POST 方法的两项工作。第一种就是 *POST-to-append*，向某个资源发送一条 POST 请求用以在该资源的下一级中创建一个新的资源。当客户端发送一个 POST-to-append 请求的时候，它会在请求的实体消息体中添加所希望创建的资源的表述信息并发送给服务器。

我在第 2 章中曾经使用 POST-to-append 来通过微博 API 发布一条微博。因为我在展示 DELETE 方法的时候已经将那条微博删除了，下面我们就创建一条新的微博：

```
POST /api/ HTTP/1.1
Content-Type: application/vnd.collection+json

{
  "template" : {
    "data" : [
      {"name" : "text", "value" : "testing"}
    ]
  }
}
```

对 POST-to-append 请求而言，最常见的响应码是 201 (Created)。它用于告知客户端一个新的资源已经创建成功。Location 报头用于告诉客户端这个新资源的 URL 地址。另一种常见的响应码是 202 (Accepted)，这表示服务器打算按照提供的表述信息来创建一个资源，但是现在还没有真正创建完成。

POST 方法既不安全也不幂等，如果我发送了 5 次 POST 请求，我会收到 5 条新的微博，这 5 条微博的 text 的值都一模一样，唯一不同的是 date_created。

这就是 POST-to-append。但是很可能你除了曾经用 POST “创建一个新的资源”之外，你还曾用它做过各种各样其他的事情。这是 POST 的另一项工作。这项工作被称为 overloaded POST，我会在本章后面的内容中进行讨论。

PUT

PUT 请求是用于修改资源状态的请求。客户端一般会通过 GET 请求获取表述，然后对其进行修改，最后再将修改后的资源表述作为 PUT 请求的负载数据 (payload) 发送回去。在本节中，我将要修改一条微博的文本信息（我想要将 text 字段的值修改成字符串 tasting 以取代之前的内容）：

```
PUT /api/qlw2e HTTP/1.1
Content-Type: application/vnd.collection+json
```

38

```
{
  "template" : {
    "data" : [
      { "name" : "text", "value" : "tasting" }
    ]
  }
}
```

服务器可以自由地拒绝一个 PUT 请求，理由可以是多种多样的，比如实体消息类的意义不够明确、实体消息类试图修改服务器认为是只读的资源状态。如果服务器决定接受一个 PUT 请求，服务器就会修改资源状态来和客户端在表述中说明的状态保持一致，修改完成之后，通常会发送 200 (OK) 或者 204 (No Content) 状态码。

PUT 和 DELETE 一样是幂等的。如果你发送 10 次同样的 PUT 请求，请求的结果和你只发送 1 次请求的结果是一样的。

如果客户端知道新资源的 URL 的话，它同样能够使用 PUT 来新建一个资源。在下面的假定的例子中，我想要发布一条新的微博，并且恰好我还知道这条新微博的 URL：

```
PUT /api/als2d3
Content-Type: application/vnd.collection+json
```

```
{
  "template" : {
    "data" : [
      { "name" : "text", "value" : "Created." }
    ]
  }
}
```

```
}  
}
```

客户端应该如何构造这个充满魔力的 URL 呢？我们将在第 4 章中对此做进一步介绍。就目前而言，我们只要注意到：即便我们用 PUT 来创建一个新的资源，PUT 也还是一个幂等的操作。如果我发送 5 次 PUT 请求，这并不会相应生成 5 条具有同样内容的微博（而 5 次 POST 请求会生成 5 条相同的微博）。

PATCH

表述的信息量可能非常大。“修改表述，然后通过 PUT 方法提交”是一个简单的规则。但是如果你只是想修改资源状态中的很小的一部分，这就可能造成极大的浪费。此外，PUT 规则还可能导致与其他修改同样文档的用户发生意外的修改冲突。如果你可以仅仅向服务器发送你想要修改的那一部分数据文档，那将是非常令人高兴的。

PATCH 方法就提供了这样的功能来允许你这么做。和将完整的表述信息通过 PUT 方法发送出去不同，你可以建立一个特别的“diff”表述，并将它作为 PATCH 请求的负载数据发送给服务器。RFC 5261 介绍了一个针对 XML 文档的 patch 格式，RFC 6902 则介绍了一种针对 JSON 文档的类似格式。

```
PATCH /my/data HTTP/1.1  
Host: example.org  
Content-Length: 326  
Content-Type: application/json-patch+json  
If-Match: "abc123"  
  
[  
  { "op": "test", "path": "/a/b/c", "value": "foo" },  
  { "op": "remove", "path": "/a/b/c" },  
  { "op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ] },  
  { "op": "replace", "path": "/a/b/c", "value": 42 },  
  { "op": "move", "from": "/a/b/c", "path": "/a/b/d" },  
  { "op": "copy", "from": "/a/b/d", "path": "/a/b/e" }  
]
```

对一个执行成功的 PATCH 请求而言，最好的响应码其实是和 PUT 以及 DELETE 请求所希望的那些响应码一样的：如果服务器想要向客户端发送数据（比如已经更新的资源表述），那么 200 (OK) 是最好的选择；而如果服务器仅仅想要表示执行已经成功，那么 204 (No Content) 就已经足够了。

PATCH 方法既不是安全的，也不能保证幂等。一个 PATCH 请求有可能结果是幂等的，这种情况下，如果你不小心对同一个文档应用了两次 patch，你可能会在第二次收到一个错误信息。但这并没有定义在相关标准中。考虑到 PATCH 的协议语义，它跟 POST 一样，

是一个不安全的操作。

需要记住的是 PATCH 方法并没有定义在 HTTP 规范中。它是最近几年（RFC5789 是在 2010 年发布的）针对 Web API 而特别设计的一个扩展方法。这也就意味着，在工具支持方面，为 PATCH 方法及其所使用的 diff 文档提供的工具不如 PUT 方法的工具支持得那样好。

LINK和UNLINK

LINK 和 UNLINK 方法用于管理资源之间的超媒体链接。为了理解这些方法，你必须首先理解超媒体和链接关系（link relation），所以我会第 11 章进行详细讨论。在这里，我先暂时仅仅展示一些简单的例子。

下面这个 UNLINK 请求是要用于删除一个故事（story）（用 <http://www.example.com/story> 进行标识）和它的作者（author）（用 <http://www.example.com/~omjennyg> 进行标识）之间的链接的：

```
UNLINK /story HTTP/1.1
Host: www.example.com
Link: <http://www.example.com/~omjennyg>;rel="author"
```

下面这是一个 LINK 请求，用于声明另一个资源（用 <http://www.example.com/~drnmilk> 标识）是这个故事资源的作者：

```
LINK /story HTTP/1.1
Host: www.example.com
Link: <http://www.example.com/~drnmilk>;rel="author"
```

LINK 和 UNLINK 都是幂等的，但不是安全的。这些方法是在互联网草案（“snell-link-method”）中定义的，在草案被批准成为 RFC 之前，对它们的工具支持会比对 PATCH 方法的支持更差。

HEAD

HEAD 方法是像 GET 方法一样的安全方法，对 HEAD 方法最好的理解就是轻量级版本的 GET。服务器应该和处理 GET 方法一样处理 HEAD 方法，但是不需要发送实体消息——只需要发送 HTTP 状态码和报头：

```
HEAD /api/ HTTP/1.1
Accept: application/vnd.collection+json

HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
```

```
ETag: "dd9b7c436ab247a7b69f355f2d57994c"
Last-Modified: Thu, 24 Jan 2013 18:40:42 GMT
Date: Thu, 24 Jan 2013 19:14:23 GMT
Connection: keep-alive
Transfer-Encoding: chunked
```

用 HEAD 来代替 GET 并不会节约任何时间（服务器还是需要生成所有的 HTTP 报头），但是它确实能够节省带宽的使用。

OPTIONS

OPTIONS 是 HTTP 的原生探索机制。一个 OPTIONS 请求的返回结果包含一个 HTTP Allow 报头，这个报头展示了这个资源所支持的所有的 HTTP 方法。下面是一个 OPTIONS 请求，这个请求是发送给那条我之前通过 PUT 请求创建的微博的：

```
OPTIONS /api/als2d3 HTTP/1.1
Host: www.youtypeitwepostit.com

200 OK
Allow: GET PUT DELETE HEAD OPTIONS
```

现在我就知道我下一步所能发送的 HTTP 请求的方法了。我可以 GET 这个资源的表述，用 PUT 来修改它，或者用 DELETE 来删除它。这个资源也支持 HEAD 和（当然的）OPTIONS，但是不能理解 PATCH 扩展以及 LINK 和 UNLINK。

OPTIONS 是一个很好的方法，但是很少有人使用它。设计良好的 API 会为 GET 请求发送超媒体文档（见第 4 章）作为响应，并用这个文档来宣传一个资源的功能。这些文档中的链接和表单阐明了客户端下一步所能发起的 HTTP 请求。而设计低劣的 API 使用人类可读的文档来说明客户端能发起哪种 HTTP 请求。

◀ 41

Overloaded POST

现在是时候揭示 HTTP 密室里的骷髅了。HTTP POST 方法有一个很肮脏的秘密。一个如果你做过 web 开发的工作就有可能遇到过的秘密。POST 不仅仅被用于创建新的资源。在我们用浏览器上网的时候，HTTP POST 也被用于传输任何形式的变化。它将 PUT、DELETE、PATCH、LINK 和 UNLINK 所有的方法混合成一个方法。

这是一个在 web 上可能会看到的一个 HTML 表单。这个表单的目的是编辑以前发布的微博：

```
<form method="POST" action="/blog/entries/123">
  <textarea>
    Original content of the blog post.
```



```
</textarea>
<input type="submit" class="edit-post" value="Edit this blog post.">
</form>
```

在协议语义的范围内，“编辑这条微博”这个操作听起来像是一个 PUT 请求。但是 HTML 表单不能触发一个 PUT 请求。HTML 数据格式并不允许这么做，所以我们使用 POST 来进行代替。

这完全是合法的。HTTP 规范中说 POST 可以用于：

向数据处理流程提供例如表单提交结果的数据块。

这个“数据处理流程”就可以无限扩展了。它可以将任何数据作为 POST 请求的一部分发送出去，不论是出于什么目的，这都是合法的。这个定义太模糊了以至于 POST 请求实际上没有任何协议语义了。此时的 POST 并不真正表示“创建一个新的资源”，而是表示“任意而为（whatever）”。

我将 POST 的这种“任意而为（whatever）”用法称为重载的（overloaded）POST。因为一个重载的 POST 请求没有协议语义，你只能在应用语义的层面上来理解它。

我会在后面的章节中讲到很多应用语义的内容，现在我仅仅要指出的是这个 HTML 表单中的应用语义。这个表单的应用语义是这个提交按钮的 CSS 类（edit-post）以及这个按钮上的可读标签（“Edit this blog post”）。

那两个字符串还是不足以说明一切。直到现在，应用语义还是知道得太少，以致于我建议一点也不要使用重载的 POST。但是如果你按照我在第 8 章中的建议执行的话，你可以使用一个配置文件来可靠地向客户端传达应用语义。但是它还是不能够像协议语义那么可靠（每个发送 GET 请求的 HTTP 客户端都知道 GET 的含义），但是我们将来会做到的。

由于重载的 POST 请求可以用来完成任何事情，所以这种 POST 方法既不是安全的也不是幂等的。某个特定的重载的 POST 可能事实上是安全的，但是从 HTTP 考虑，POST 方法是不安全的。

应该使用哪些方法？

一个 REST 系统是由各个独立的组件组合而成的：服务器、客户端、缓存、代理、缓存代理等。这些组件是由不同的人建立的，并且在开始会话之前互相是不知道对方的存在的，它们只能通过 HTTP（或者类似的协议）相互传递文档来进行通信。在进行通信之前，所有的组件都要在协议语义上达成一致，这是必需的，否则这些组件就不能理解对方的含义了。

HTTP 的协议语义很大程度上是通过 HTTP 方法定义的。但是这些方法之间还有很多的冗余。PUT 可以代替 PATCH, GET 方法可以完成 HEAD 的工作, POST 可以代替任何方法。我们真的需要所有的这些方法吗?

现实中, 并不存在官方的协议语义集合。我们可以看到和听到很多关于“哪些 HTTP 方法是最好的”的好笑的争论, 但是这归根到底都是某些社区成员提起的。当你选择了某些你想要使用的 HTTP 方法的时候, 你也就选择了某个社区: 一个理解这些方法的客户端及其他组件的社区。

对大部分的 web API 而言, 我建议使用的方法有 GET、POST、PUT、DELETE 和 PATCH。但是我也记得我曾经在很多情况下建议过不同的方法:

- 在 2008 年之前, PATCH 方法还不存在。那时候, 我对 web API 所建议的方法集合是 GET、POST、PUT 和 DELETE。
- 1997 年, HTTP 1.1 (RFC 2068) 规范的第一个版本定义了 LINK 和 UNLINK 方法。但是到了 1999 年, 这两个方法在最终规范 (RFC 2616) 中被删除了, 因为在当时并没有人使用这两个方法。
LINK 和 UNLINK 作为 HTTP 正式的协议语义的一部分的状态只维持了短短两年的时间。然后它们很快就被删除了。由于这些方法对很多的 API 而言是很有帮助的, 所以互联网草案 “snell-link-method” 又试图将它们重新添加回来。
- WebDAV 标准 (由 RFC 4981 规定, 我将在第 11 章进行简要介绍) 定义了 7 个新的 HTTP 方法用于让 API 处理文件系统中的文件资源。这些方法包括有 COPY、MOVE 和 LOCK。
- 当人们在通过 web 浏览器上网的时候, 我们完全忽略了 HTTP 规范中定义的大部分方法, 而仅仅使用 GET 和 POST 就可以顺利操作。这是由于 HTML 文档的协议语义仅仅支持 GET 和 POST。
- CoAP 协议 (见第 13 章) 也定义了 GET、POST、PUT 和 DELETE 方法。这些方法是根据 HTTP 方法命名的, 但是它们的含义略有不同, 因为 CoAP 毕竟不是 HTTP。

如果你想要一个完全由 HTML 文档描述的 API, 那么你的协议语义将被局限于 GET 和 POST。如果你想要和类似于 Web Folders (微软公司研发) 这样的文件系统 GUI 程序通信, 你就要使用 HTTP 和 WebDAV 扩展了。如果你想要同各式各样的 HTTP 缓存和代理进行会话, 你就应该远离 PATCH 方法, 远离 RFC2616 中没有定义的其他方法。

在 Web API 的世界里, 有的社区比较大, 有的社区比较小。如果你离开大路而去创造你自己的协议语义, 你就会把自己孤立于一个只有你一个人的社区。

超媒体

到目前为止，整个故事是这样的：URL 标识出资源，客户端向这些 URL 发起一系列 HTTP 请求，然后服务器在响应中向客户端发送表述。随着时间的推移，客户端通过这些表述建立起了一个资源状态的全景图。最后，客户端发起一个意义重大的 PUT、POST 或者 PATCH 请求，将一个表述发送回服务器从而更改资源的状态。

更近一步看，你将会发现还有很多问题尚未解答：客户端如何知道应该发起哪种请求？互联网上存在着那么多的 URL，客户端怎么知道哪些 URL 背后存在着表述，而哪些将会返回给你一个 404 错误？客户端应该在它的 POST 请求中发送实体消息体吗？如果应该，那实体消息体应该是什么样的？HTTP 定义了一系列的协议语义，但是在当前 URL 所对应的服务器上又支持这些语义中的哪些呢？

解开这些谜题所缺失的那块拼图便是超媒体。超媒体将资源互相连接起来，并以机器可读的方式来描述它们的能力。通过合理地使用超媒体，便可以解决或至少是改善现今 web API 存在的可用性和稳定性问题。

与 REST 类似，超媒体并不是由某个标准文档所描述的一种单一的技术。超媒体是一种策略，并且可以由多种技术以不同的方式来实现。我将会在接下来的 3 章中讨论到数个超媒体标准，而更多标准将留待第 10 章讨论。不过在实际工作中，一切还得取决于哪些技术可以满足你的业务需求。

超媒体策略通常都拥有同样的目标。超媒体是服务器用以和客户端进行对话的一种方式，客户端从而可以知道未来将可以向服务器发起什么样的请求。它就是一个由服务器提供的菜单，客户端可以从中自由地进行选择。服务器通常都知道可能发生哪些事，但是客户端将决定实际发生什么。

其实这并不是什么新鲜的话题，我们的万维网就是以这种方式工作的，并且我们都想当然地认为它就应该是这样工作的，其他的任何方式都是一种无用的历史的倒退。但是在 API 的世界里，超媒体仍然是一个令人难以理解且富有争议的话题。这也说明了为什么如今的 API 在应对变化时还是显得如此糟糕。

在这一章里，我将揭开超媒体神秘的面纱，从而可以让你为 Web 创建出更具灵活性的 API。

将HTML作为超媒体格式

你应该已经对 HTML^{注1} 非常熟悉，所以我们从一个 HTML 实例开始讲起，下面是一个 HTML<a> 标签：

```
<a href="http://www.youtypeitwepostit.com/messages/">a
  See the latest messages
</a>
```

该标签是一个简单的超媒体控件。它是一段对 HTTP 请求的描述，用于描述浏览器在将来可以发起的一种 HTTP 请求。<a> 标签对浏览器来说就像是一种暗号，暗示它可以发起一个如下的 HTTP GET 请求：

```
GET /messages HTTP/1.1
Host: www.youtypeitwepostit.com
```

HTML 标准中指出，当用户激活一个链接时，该用户将会“访问”到该链接另一端的资源^{注2}。而实际情况是，这表示获取该资源的一个表述并将它在浏览器窗口中展现出来，从而替换掉了原来的表述（即包含该链接的那个页面）。当然，这并不是自动发生的。在你单击该链接前什么都不会发生。每个 <a> 标签都是 web 服务器的一个承诺，它确保每个确定的 URL 都命名了一个你可以访问到的资源。如果你向自己随意构造的一个 URL 发送 GET 请求，例如：`http://www.youtypeitwepostit.com/give-me-the-messages?please=true`，你将可能仅仅得到一个 404 错误。

相比于 <a> 标签，另一个 HTML 的超媒体控件 标签如下：

```

```

 标签同样描述了浏览器在不久的将来可以发起的一种 HTTP 请求，但是它并不会导致你从一个文档转移到另一个。相反，被链接资源的表述应该会以图片的形式嵌入到

注1 有两个 HTML 规范你应该需要了解：HTML 4 规范和 HTML 5 规范。它们都是由 W3C 制定的开发标准。HTML 4 已经稳定存在了超过十年的时间；HTML 5 的制定工作则还在进行中。

注2 见 HTML 4 规范的 12.1.1 章节。

当前的文档中。当你的浏览器发现一个 `` 标签时，它并不会提示你单击任何事物，而是自动向该图片发起一个请求。然后它会将该图片的表述整合到你当前浏览的文档中，这个过程同样不需要你的许可。

让我们来看一个更加复杂的超媒体控件——HTML 表单：

```
<form action="http://www.youtypeitwepostit.com/messages" method="post">
  <input type="text" name="message" value="" required="true" />
  <input type="submit" value="Post" />
</form>
```

这个表单描述了一个目标 URL 为 `http://www.youtypeitwepostit.com/messages/` 的请求。这和我在 `<a>` 标签中所使用的 URL 是同一个，但是 `<a>` 标签描述的是一个 GET 请求，而该表单描述的是一个 POST 请求。

这个表单不仅仅给你提供了一个 URL，它还会帮你发起一个 POST 请求。这里还有两个控件——一个 text 文本框和一个 submit 提交按钮，它们都在浏览器中被渲染成了 GUI 元素。

当你单击提交按钮时，你在文本框中输入的值和按钮本身的 `value` 属性值都会被浏览器根据 HTML 规范中指定的规则集转化为一个表述。这些规则集指定了该表述的媒体类型将会是 `application/x-www-form-urlencoded`，它看上去是这样的：

```
message=Hello%21&submit=Post
```

将上面的内容整合在一起，该 `<form>` 标签告诉你的浏览器，它可以发起一个如下的 POST 请求：

```
POST /messages HTTP/1.1
Host: www.youtypeitwepostit.com
Content-Type: application/x-www-form-urlencoded

message=Hello%21&submit=Post
```

它和 `<a>` 标签一样，服务器可以通过它来对你进行引导，只是形式上没有这么强硬。如果你不想填写这个表单，你完全可以忽略它。但是如果你愿意填写这个表单，你可以在 `message` 字段中填写任何你想要的内容（虽然服务器有可能会拒绝掉某些值）。服务器通过 `<form>` 标签告诉你，在所有你可能发起的 POST 请求中，有一个请求类型有可能会产生一些有用的东西。这就是向 `/messages` 发起的 POST 请求，它包含了一个经由表单编码（form-encoded）方式编码过的实体消息体，这个消息体含有 `message` 的值。

我们再讨论一个 `<form>` 标签：

```
<form method="GET" action="http://www.youtypewepostit.com/messages/">
  <input type="text" id="query" name="query"/>
  <input type="submit" name="Search"/>
</form>
```

这个表单也拥有一个你可以向其填写内容的 text 输入框，但是该表单同时也告诉你它将发起的是一个 GET 请求，而 GET 请求将不会包含实体消息体。取而代之的是，你输入文本框的数据将会整合到请求的 URL 中——同样地，这也是根据 HTML 规范中指定的规则来拼接的。

如果你填写完了这个表单，你的浏览器将会发起如下的 HTTP 请求：

```
GET /messages/?query=rest HTTP/1.1
Host: www.youtypeitwepostit.com
```

做一下总结，常见的 HTML 控件允许服务器可以向客户端描述如下 4 种类型的 HTTP 请求。

- `<a>` 标签描述了一个针对特定 URL 的 GET 请求，该请求只会在用户触发控件时产生。
- `` 标签描述了一个针对特定 URL 的 GET 请求，它将会在后台自动发起。
- 拥有 `method="POST"` 属性的 `<form>` 标签描述了一个针对特定 URL 的 POST 请求，该请求将会拥有一个由客户端构造的实体消息体。该请求只会在用户触发控件时产生。
- 拥有 `method="GET"` 属性的 `<form>` 标签描述了一个针对由客户端构造的自定义 URL 的 GET 请求。该请求只会在用户触发控件时产生。

HTML 同样也定义了很多其他稀奇古怪的超媒体控件，还有其他一些同样很生僻的可以用来定义控件的数据格式。所有这些都归入到由 Fielding 论文给出的正式的超媒体定义中：

超媒体是由应用控制信息来定义的，而这些控制信息或内嵌在信息表达（presentation of information）之中，或作为上层凌驾于信息表达之上。

万维网到处充斥着 HTML 文档，而这些文档描述的又全都是人们喜欢阅读的事物——报价、统计数字、个人信息、散文和诗歌。但是所有这些事物都被归入到信息表述的范畴。在信息表述的概念里，Web 与一本印刷出版的图书没有太多区别。

是应用控制信息将一个 HTML 文档与图书区别开来。我现在就在讨论超媒体控件，也就是那些人们一直在与之交互但是又很少更进一步地进行审查的东西。`` 标签告诉浏览器嵌入某些图片，`<a>` 标签将最终用户传送到 Web 的另一处，而 `<script>` 标签则为

浏览器提供 JavaScript 以供执行。

一个包含有一首诗歌的 HTML 文档或许会拥有一个链接来指向“该作者的其他诗歌”，或者是一个让读者可以“为该诗歌评分”的表单。这便是无法在印刷出版的诗歌图书中所展现的应用控制信息。应用控制信息的存在或多或少会减弱一首诗歌在情感上所产生的影响，但是只包含一首诗歌文本内容的 HTML 文档将不算是 Web 的全面参与者，它只是在模拟一本印刷出版的图书。

URI 模板

你使用 HTML 的 `<form>` 标签而创建的自定义 URL 只能局限在表单中。`http://www.youtypeitwepostit.com/messages/?search=rest` 这个 URL 看上去并不美观。但从技术的层面来说，这并不要紧。URL 并不需要看上去很美观，甚至不需要被人类的眼睛所理解。但是我们人类偏好美观的 URL，比如 `http://www.youtypeitwepostit.com/search/rest`。

HTML 中的超媒体控件无法告诉浏览器如何来构造一个像 `http://www.youtypeitwepostit.com/search/rest` 这样的 URL。但是另一种超媒体技术——URI 模板可以做到这一点。URI 模板由 RFC 6570 定义，它们看上去是这样的：

```
http://www.youtypeitwepostit.com/search/{search}
```

这并不是一个合法的 URL，因为它含有花括号。通过这些花括号，我们可以识别出这个字符串是一个 URI 模板。RFC 6570 中说明了你可以如何将这个字符串转化为无数的 URL。它规定你可以将 `{search}` 替换为任何你想要的内容，只要该字符串在 URL 内是合法内容。

- `http://www.youtypeitwepostit.com/search/rest`
- `http://www.youtypeitwepostit.com/search/RESTful%20Web%20APIs`

该 HTML 表单：

```
<form method="GET" action="http://www.youtypeitwepostit.com/messages/">
  <input type="text" id="query" name="query"/>
  <input type="submit" name="Search"/>
</form>
```

几乎等价于以下的 URI 模板：

```
http://www.youtypeitwepostit.com/messages/?query={query}
```

这是一种非常常见的情况，因此 URI 模板标准为带有查询字符串的 URL 定义了一种简写方式。以下的 URI 模板与前一个模板完全是等价的，同样也等价于前面的 HTML 表单：


```
http://www.youtypeitwepostit.com/messages/{?query}
```

URI 模板标准有着非常丰富的实例，以下是一些范例模板以及通过这些模板可以获得的一些 URL：

如果参数值按如下设置：

```
var    := "title"
hello  := "Hello World!"
path   := "/foo/bar"
```

那么这些 URI 模板：

```
http://www.example.org/greeting?g={+hello}
http://www.example.org{+path}/status
http://www.example.org/document#{+var}
```

将扩展为以下的 URL：

```
http://www.example.org/greeting?g=Hello%20World!
http://www.example.org/foo/bar/status
http://www.example.org/document#title
```

虽然，相比于 HTML GET 表单，URI 模板显得更加简短和灵活，但是这两种技术差别并不是很大。URI 模板和 HTML 表单都允许 web 服务器通过简短的字符串来描述无穷多的 URL。而 HTTP 客户端可以从无穷多的 URL 家族中选取一个，通过插入一些值来发起一个特定 URL 的 GET 请求。

URI 模板本身并没有意义，一个 URI 模板需要内嵌到一种超媒体格式中。这个概念是指每个需要该功能的标准应该只需要使用 URI 模板，而非像在 RFC 6570 发布前那个时代，都需要再制定一种自定义格式。

URI vs URL

我一直在尽可能延后这个话题的讨论，不过现在已经是时候来说明 URL（我几乎在本书的每个地方都在使用这个词）和 URI（更加一般化的一个词且常被用于技术名称中，比如 URI 模板）之间的区别了。大部分的 web API 都只专门处理 URL，本书中的 web API 也是如此。对本书大部分内容而言，它们之间的区别并不要紧，但是一旦重要起来（将会在第 12 章中出现这样的场景），也可以说是非常重要的。

URL 是一个用以标识资源的简短字符串，URI 也是一个用以标识资源的简短字符串，而每个 URL 都是一个 URI。它们都在同一个标准：RFC 3986 中进行描述。

那它们之间的区别到底是什么呢？就本书关注的主题而言：一个 URI 并不保证具有表述。

一个 URI 什么都不是，它只是一个标识。而一个 URL 是一个可以被引用（dereferenced）的标识。这就是说，一台计算机可以以某种方式得到一个 URL 从而获取其背后的资源。

如果你看到 `http:URI`，你就已经知道计算机如何从该目标获取表述：通过发起一个 HTTP GET 请求。如果你看到得是 `ftp:URI`，你也知道计算机如何从该目标获取表述：通过启动一个 FTP 客户端并执行某些 FTP 命令。这些 URI 其实都是 URL。它们都拥有与之关联的协议：即从这些资源获取表述的规则（指导计算机遵循的非常具体的规则）。

这是一个非 URL 的 URI：`urn:isbn:9781449358063`。它也指向一个资源：该图书的一个印刷版本。它不是该图书特定的复制品，而是该完整版本的一个抽象概念（请记住，一个资源可以是任何事物）。这个 URI 并不是 URL，因为……它的协议是什么？计算机将如何从它获取表述？你做不到。

如果没有了 URL，你将无法获取表述。如果没有了表述，也就没有了表述性状态移交。如果一个资源没有使用 URL 来标识，该资源将无法满足不同项 Fielding 约束。它同样也无法满足自描述信息的约束，因为它无法发送任何消息。一个表述可以链接到 URI 而非 URL（``），但是这并不满足超媒体约束，因为客户端将无法访问该链接。

51

这里有一个标识了图书印刷版的 URL：`http://shop.oreilly.com/product/0636920028468.do`。你可以向该 URL 发送一个 GET 请求并获取到该版本的一个表述。获取到的并不是该书的实体复制品，而是一个表达了该资源某些状态的 HTML 文档：标题、书的页码数等。该 HTML 文档同样也包含有超媒体，比如该书作者的链接——并不是其人本身，而是关于他们的一些信息。由 URL 标识的资源将满足所有 Fielding 约束。

出于很多原因，我们需要使用 URI 而非 URL，我将会在第 12 章讨论资源描述策略时涉及到这些场景，但是这种情况相当罕见。通常，当你的 web API 在引用资源时，它都应该使用 http 或 https 模式的 URL，而这样的 URL 才能正常运作：它将可以在 GET 请求的响应中发送一个有用的表述。

Link 报头

有这样一项技术，它可以将超媒体设置到你想象不到的地方：将超媒体放入一个 HTTP 请求或响应的报头中。RFC 5988 定义了 HTTP 的这项扩展，一个称为 Link 的报头。该报头让你可以为通常不支持超媒体的实体消息体（比如 JSON 对象和二进制图片）添加简单的超媒体控件。

下面是一个故事的普通文本表述，该故事被拆分成多个连续的部分（该 HTTP 响应的实

体消息体包含了该故事的第一部分，而 Link 报头指向了第二部分）：

```
HTTP/1.1 200 OK
Content-Type: text/plain
Link: <http://www.example.com/story/part2>;rel="next"
It was a dark and stormy night. Suddenly, a...
(剩余部分将在 part 2 中继续)
```

该 Link 报头的功能近似于 HTML `<a>` 标签。我建议在任何可能的情况下使用真正的超媒体格式，但是在无法选择它们的场景中，Link 报头将会非常有用。

HTTP 中的 LINK 和 UNLINK 扩展方法使用了 Link 报头。你现在应该可以理解这个来自第 3 章的例子的含义了：

52

```
LINK /story HTTP/1.1
Host: www.example.com
Link: <http://www.example.com/~drmlk>;rel="author"
```

超媒体的作用

我将在本书中提到很多的超媒体数据格式，但是只是一味地逐一向你介绍这些技术的话，恐怕对你并没有太大的帮助。我们需要退回来首先看看超媒体到底是用来做什么的。

超媒体控件承担了 3 项工作：

- 它们告诉客户端如何构建 HTTP 请求：使用什么 HTTP 方法，使用什么 URL，发送怎样的 HTTP 报头和 / 或实体消息体。
- 它们对 HTTP 响应做出承诺，给出建议的状态码、HTTP 报头以及服务器有可能在请求的响应中发送的数据。
- 它们给出客户端应该如何将响应集成到其工作流的建议。由于 HTML GET 表单和 URI 模板所完成的工作是相同的，所以让人们觉得它们很相似。它们都可以告诉客户端如何构造一个在 HTTP GET 请求中使用的 URL。

引导请求

一个 HTTP 请求具有 4 个部分：方法、目标 URL、HTTP 报头和实体消息体。超媒体控件可以引导客户端来指定所有这 4 个部分。HTML `<a>` 标签可同时指定请求中所使用的目标 URL 和 HTTP 方法：

```
<a href="http://www.example.com/">An outbound link</a>
```

目标 URL 被显式地定义在 href 属性中，而 HTTP 方法也被显式地定义了：HTML 规范

指出当一个最终用户单击链接时，<a> 标签将转变成一个 GET 请求。下面的 HTML 表单定义了一个将来可能发起的 HTTP 请求的方法、目标 URL 以及实体消息体。

```
<form action="/stores" method="get">
  <input type="text" name="storeName" value="" />
  <input type="text" name="nearbyCity" value="" />
  <input type="submit" value="Lookup" />
</form>
```

HTTP 方法和目标 URL 都被显式地定义了。实体消息体也按照一系列针对客户端的问题进行了定义。客户端需要指定它想赋给变量 `storeName` 和 `nearbyCity` 的值。如此客户端便可以构建一个经过表单编码 (form-encoded) 的服务器可以接受的实体消息体 (谁说一定要经过表单编码的? 这是在 HTML 规范中 <form> 标签处理规则中明确定义的)。

下面的 URI 模板仅仅指定了一个 HTTP 请求的目标 URL，其他部分并没有指定：

```
http://www.youtypeitwepostit.com/messages/{?search}
```

该目标 URL 取决于那个有待填充的变量值，就像 HTML 表单产生的实体消息体取决于客户端赋予的变量值一样。客户端将使用某种算法来将 URI 模板和它的 `search` 变量的期待值转化为一个真正的 URL，比如：`http://www.youtypeitwepostit.com/messages/?search=rest`。

一个 URI 模板仅仅定义了 HTTP 请求的目标 URI。它并不会告诉你应该发起 GET 请求，还是 POST 请求，或者是任何特定类型的请求。这就是我说 URI 模板本身并没有意义以及需要将它们绑定到另一种超媒体技术上的原因。

下面的 HTML 表单告诉客户端将为 HTTP 的 Content-Type 报头设置一个特定值：

```
<form action="POST" enctype="text/plain">
  ...
</form>
```

通常，HTML POST 表单产生的实体消息体都是经过表单编码的，它们在被发送到网络时，请求的 Content-Type 报头都会被设置为 `application/x-www-form-urlencoded`。但是只要指定 <form> 标签的 `enctype` 属性，便可以覆盖这种默认的行为。一个拥有属性 `enctype="text/plain"` 的表单将会告诉浏览器使用普通文本格式来编码它的实体消息体，并在将其发送到网络时将 Content-Type 报头设置为 `text/plain`。

这并不是一个完美的例子，因为 `enctype` 属性只会改变 Content-Type 报头的值，它只能起到一种改变实体消息体的辅助作用。但是这是我在使用像 HTML 这样流行的超媒体格式时能想到的最好的例子了。

超媒体控件通常都允许 HTTP 客户端自由发送任何它们希望的报头，但是这种放任的态度仅仅是一种约定俗成。一个超媒体控件可以非常具体地描述一个 HTTP 请求。它可以指导客户端使用指定的 HTTP 方法，按照指定的规则构建实体消息体，提供指定的 HTTP 报头值，进而向指定的 URL 发送一个 HTTP 请求。

对响应做出承诺

下面是另一个 HTML 标签：

```

```

像 `<a>` 标签一样，一个 `` 标签将会承诺客户端可以向特定的 URL 发起一个 GET 请求。但是 `` 标签还做出了另一个承诺：服务器将会在 GET 的响应中发送某种图片的表述。

下面是另一个例子——一个来自 Atom 发布协议（Atom Publishing Protocol，我将会在第 6 章对该协议进行详细讨论）的简单 XML 超媒体控件：

```
<link rel="edit" href="http://example.org/posts/1"/>
```

这看上去足够简单；事实上，这个 `<link>` 标签可以合法地展示在一个 HTML 文档中，但是需要根据 AtomPub 标准来解释该标签，这个 `rel="edit"` 给你提供了关于 `http://example.org/posts/1` 处的资源的足够多的信息。

首先，`rel="edit"` 告诉我们在 `http://example.org/posts/1` 处的资源支持 PUT、DELETE 以及 GET 方法。你可以通过 GET 获取该资源的一个表述，然后修改该表述，通过 PUT 将它提交回服务端来改变资源的状态。这是使用 HTTP 的一个完美的典范，很多事情无须显式地声明。但鉴于大多数的 HTTP 资源都不会对 PUT 和 DELETE 做出响应，这就需要花费时间来进行沟通。

更重要的是，`rel="edit"` 意味着客户端无须再猜测向 `http://example.org/posts/1` 发送 GET 请求后会收到什么类型的表述。你将会收到在 AtomPub 中被称为成员入口（Member Entry）类型的文档（具体的细节现在并不重要——如果你想了解更多关于 AtomPub 的内容请直接跳到第 6 章）。

服务器向客户端做出了承诺：如果你发起一个 GET 请求，你将会接收到一个 AtomPub 成员入口的表述作为回报。客户端无须盲目地发送 GET 请求而只为一探 Content-Type 内容的究竟。客户端可以预知表述将会是 `application/atom+xml` 类型，同时客户端也可以知道有关该表述的应用语义。

workflow control

超媒体的第三件工作便是表述资源之间的关系。下面是一个最适合解释这一点的例子，这是一个 HTML<a> 标签：

```
<a href="http://www.example.com/">An outbound link</a>
```

如果你在 web 浏览器中单击这个链接，你的浏览器的当前页面将会跳转到该链接 href 属性指定的 web 页面。而老的页面将会被废弃，最终将只会成为你浏览器历史记录中的一条记录。这个 <a> 标签是一个转出（outbound）的链接：当一个超媒体控件被激活时，将会以全新的状态来替换客户端当前的应用状态。

相比于 <a> 标签，我们来看看 HTML 中的 标签：

```

```

这也是一个链接，但是它不是一个转出的链接；这是一个内嵌的链接。内嵌的链接将不会替换客户端当前的应用状态，内嵌的链接将会增强当前的页面。如果你访问了一个 HTML 中包含 标签的网页，该图片将会在一个独立的 HTTP 请求中自动加载（无

55

须你进行单击操作），并且展示在同一窗口的当前网页中。

一个 HTML 文档不仅仅可以嵌入图片，下面的 HTML 标签将会下载并运行一些可执行的 JavaScript 脚本：

```
<script type="application/javascript" src="/my_javascript_application.js"/>
```

以下的标签将会下载一个 CSS 样式表并应用于当前的整个文档：

```
<link rel="stylesheet" type="text/css" href="/my_stylesheet.css"/>
```

以下的标签将会在当前页面中嵌入另一个完整的 HTML 文档：

```
<frameset>
  <iframe src="/another-document.html" />
</frameset>
```

所有以上这些都是嵌入性的链接，将一个文档嵌入另一个文档中的过程也叫作内嵌（transclusion）。

当然，客户端可以自由选择是否忽略服务器的引导。有一些浏览器的扩展可以阻止浏览器引用 <script> 标签中所引用的文件，还有一些用于增强可读性的浏览器配置选项会覆盖由样式表指定的格式指令。前面讲到的这些标签与 <form> 标签一样，都是给予客户端一些暗示，告诉它哪些请求可以给客户端提供所需要的内容，但是客户端通常也可以自由决定不发起任何请求。

当心冒牌的超媒体！

市场上存在着很多由深谙超媒体益处的开发者所设计的 API，但是严格说来它们并不包含超媒体。想象下面是由一个书店 API 提供的一份 JSON 表述：

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "title": "Example: A Novel",
  "description": "http://www.example.com/"
}
```

这是一本图书的表述，其中 `description` 字段的内容看上去是一个 URL：`http://www.example.com/`。但是这算是一个链接吗？可以认为 `description` 链接到一个提供了描述信息的资源吗？又或者这就是一段文本描述，而一些自作聪明的人输入了一些文本正好是一个有效的 URL 格式？

56

正式地说，`http://www.example.com/` 就是一段字符串。而 `application/json` 这种媒体类型并没有定义任何超媒体控件，所以即使表述的某一部分看上去很像一个超媒体链接，但事实上它并不是，那只是一段字符串。

如果你打算尝试使用这样一个 API，你将不会很教条地否定这些链接的存在。相反，你将会阅读某些由 API 提供者编写的人类可读的文档。这些文档将会做出一些约定说明，约定那些 API 提供者如何在不支持超媒体的格式（JSON）中嵌入超媒体链接。这样一来你便可以知道如何来区分链接和字符串，并且你将可以编写客户端程序来发现和访问这些超媒体链接。

但是你的客户端程序将只能为该特定的 API 工作，而你所阅读的文档也只是一份一次性的 *fiat* 标准。你所使用的下一个 API 将会有另一套截然不同的、用以在 JSON 中嵌入超媒体链接的约定，而你将不得不再次重复你做过的这些工作。

这就是 API 设计者不应该设计提供普通 JSON 的 API 的原因。你应该使用一种真正支持超媒体的媒体类型，你的用户将会为此而对你表示万分感谢。他们将可以使用现有的针对这类媒体类型的代码库，而不是为你的 API 重新造轮子。

JSON 在相当长的时间里一直都是 API 中最流行的表述格式，但是直到几年前，都还并不存在基于 JSON 的超媒体格式。正如你将在接下去几章中所看到的，这些已经发生了变化。请不要为你必须放弃 JSON 而担心，你即将拥有真正的超媒体。

语义挑战：我们该怎么做？

在第 1 章的末尾，我提出了一项挑战：“我们该如何编写程序让计算机来决定单击哪个链接呢？”web 浏览器的工作方式是将表述发送给人类，并由人类来做出所有的决定。如果我们在每个步骤中去除掉与人类的协商，我们又如何才能完成类似的行为呢？

向客户端提供链接便是我们在正确方向上走出的第一步。在无穷多的合法 HTTP 请求集中，超媒体文档将会马上帮你指出该站点中的哪些请求将是有用的，因此客户端无须再为此进行猜测。

但是这并不够，假设一个 HTML 文档只含有两个链接：A 和 B。客户端将可以发起两个可能的请求，那么客户端该如何选择呢？它将基于什么依据来进行决断呢？

好吧，假设这两个链接中有一个由 HTML `` 标签进行表述，而另一个由 `<script>` 标签进行表述。就 HTTP 协议而言，这两个链接本身并没有什么差别。它们都拥有相同的协议语义，它们都将分别触发一个 GET 请求，而这个请求发向预先指定好的 URL，但是这两个链接拥有不同的应用语义。在 `` 标签另一端的表述将被作为一张图片进行展示，而 `<script>` 标签另一端的表述将被作为客户端代码而被执行。

57

对于某些客户端而言，这些信息已足够帮它做出决策。一个被设计用于抓取某网页所有图片的客户端将会访问 `` 标签的链接而忽略掉 `<script>` 标签的链接。

上面的例子展示了超媒体控件可以为语义鸿沟架起桥梁。它们可以告诉客户端为什么要发起某个 HTTP 请求。

但是对于大多数客户端来说，`` 标签和 `<script>` 标签之间的区别还不足以帮助它们做出决策。“图片”和“脚本”都是非常一般的应用语义。由 HTML 描述的应用便是万维网，这是一个可用于各行各业的非常灵活的应用。

每当我考虑到应用语义的时候，我通常都会在一个更高的层次进行思考。我会区分 wiki 和在线商店之间概念上的差别。它们都是网站，它们都使用内嵌的图片和脚本，但是它们表示着不同的事物。

一种超媒体格式不需要像 HTML 那样通用，但是它可以将 wiki 或网店应用的语义定义得足够详尽。在下一章中，我将讨论那些被设计用于表述某个特定类型的问题的超媒体格式。如果将它们抽离到问题领域之外，它们将没有任何价值。但是在这些超媒体格式的领域范围内，它们将可以非常好地应对语义挑战。

领域特定设计

在本章中，我将会选择一个问题空间（problem space）并实现一个用于展示该问题空间的 web API。关于问题空间的具体细节其实并不重要。因为它们所需要的技术都是相同的。所以我打算选择一个我能想到的最无聊的例子：迷宫游戏。

图 5-1 展示了一个只有一个入口和一个出口的简单的迷宫，我的服务器的工作就是创造出这样的迷宫并展示给客户端。

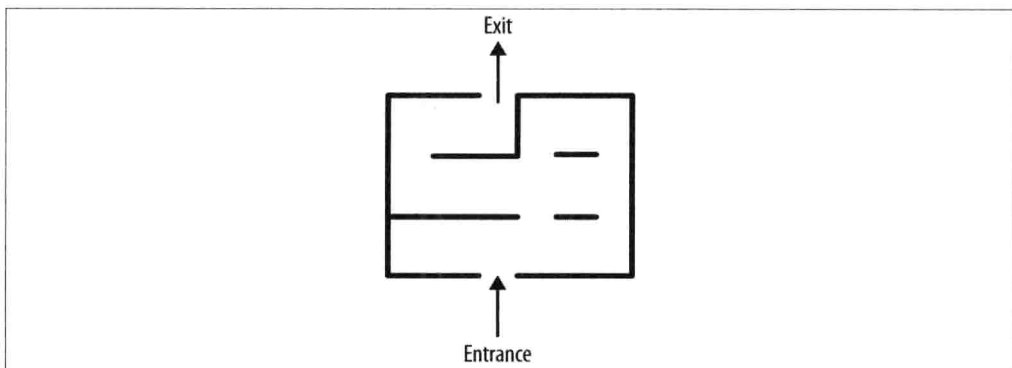


图5-1 迷宫实例（俯视图）

尽管这是一个很无聊的例子，但是对于超媒体应用程序来说，迷宫通常是一个很好的比喻。任何复杂的问题都可以表述为一个客户端必须访问的超媒体迷宫（hypermedia maze）。如果你有过以下的经历，比如曾经从杂乱无章的电话通讯录中寻找相应的联系方式，又或者你在网上商店搜索产品然后从搜索结果中购买了某些产品，你其实就已经访问过超媒体迷宫了。

我曾经见过各种各样的超媒体 API，比如用于修改复杂的保险单的 API、从目录中选择产品然后进行支付的 API 以及描述电话通讯录的 API（见第 10 章 VoiceXML）。所有的这些 API 都和我前面所展示的迷宫游戏一样：

- 问题太复杂以至于不能一次性全部理解，所以需要将问题分割为几个步骤。
- 每个客户端都是从同样的起始步骤开始处理流程的。
- 过程中的每一步，服务器都会给客户端提供若干接下来可能采取的步骤供其从中选择一个作为下一个步骤。
- 每一步，客户端都要自行决定下一步要执行的步骤。
- 客户端知道执行成功的标志是什么以及什么时候结束。

随着本书的深入，我所要处理的问题域也将更加具体。要处理的文档以及接下来可能采取的步骤也将更加复杂，但是这种按部就班的解决问题的算法是相通的，都是同样可以工作的。

Maze+XML：领域特定设计

让我们再看一下图 5-1。那是一个迷宫的图形化表述。这张图片对人类而言还有一种直观上的意义，但是对于计算机而言，计算机只有通过机器视觉算法（machine-vision algorithm）才能理解它。我们如何才能以一种便于计算机理解的方式来展示迷宫的样子呢？

解决方案有很多种，但是为了避免从零开始设计一套解决方案，我计划复用一些已经完成的工作。这里有一个称为 Maze+XML (<http://amundsen.com/media-types/maze/>) 的个人标准，我们可以通过它来以一种机器可读的格式展示迷宫。

Maze+XML 文档的媒体类型是 `application/vnd.amundsen.maze+xml`。如果你发起一个 HTTP 请求，并发现返回结果的 Content-Type 报头使用了这个字符串的话，你就会知道你现在需要一份 Maze+XML 规范说明书来全面理解收到的实体消息体了。一个领域特定的设计就是这样消除语义挑战的：定义一个用于描述问题（比如迷宫的布局）的文档格式，将这个文档格式注册为一个媒体类型，这样客户端就可以在遇到这个问题的时候立刻意识到该问题的发生了。

通常来说，我不建议创建一个新的领域特定的媒体类型。将我们的应用语义添加到一种通用的超媒体类型（我会在后面的两章中对该技术进行介绍）通常可以减少很大的工作量。如果你要着手开始某个领域特定的设计工作，你很可能最终设计出的是一个没有充分利用前人的工作的 fiat 标准。你可能不会拥有那些折磨了当今很多 API 的灵活性问题，但是你也做了更多的毫无价值的工作。

但是大多数的开发人员在设计 API 的时候，他们的第一直觉还是领域特定设计。有什么比简单解决手上的问题还自然的事情吗？这也就是我首先介绍领域特定设计的原因。展示一个能够消除语义鸿沟的自定义的超媒体格式还是比较容易的。

Maze+XML是如何工作的

我们不再像图 5-1 那样从上往下看迷宫，想象一下我们就身处迷宫内部。这时，你看到的就不是迷宫的全貌了，你能看到的仅仅是你身边周围的事物。从迷宫的入口进入，你会看到如图 5-2 所示的一样的场景：你的面前是一堵墙，你的背后是迷宫的入口。你现在有两个选择：向左走还是向右走。你没有办法知道哪个方向通向出口。

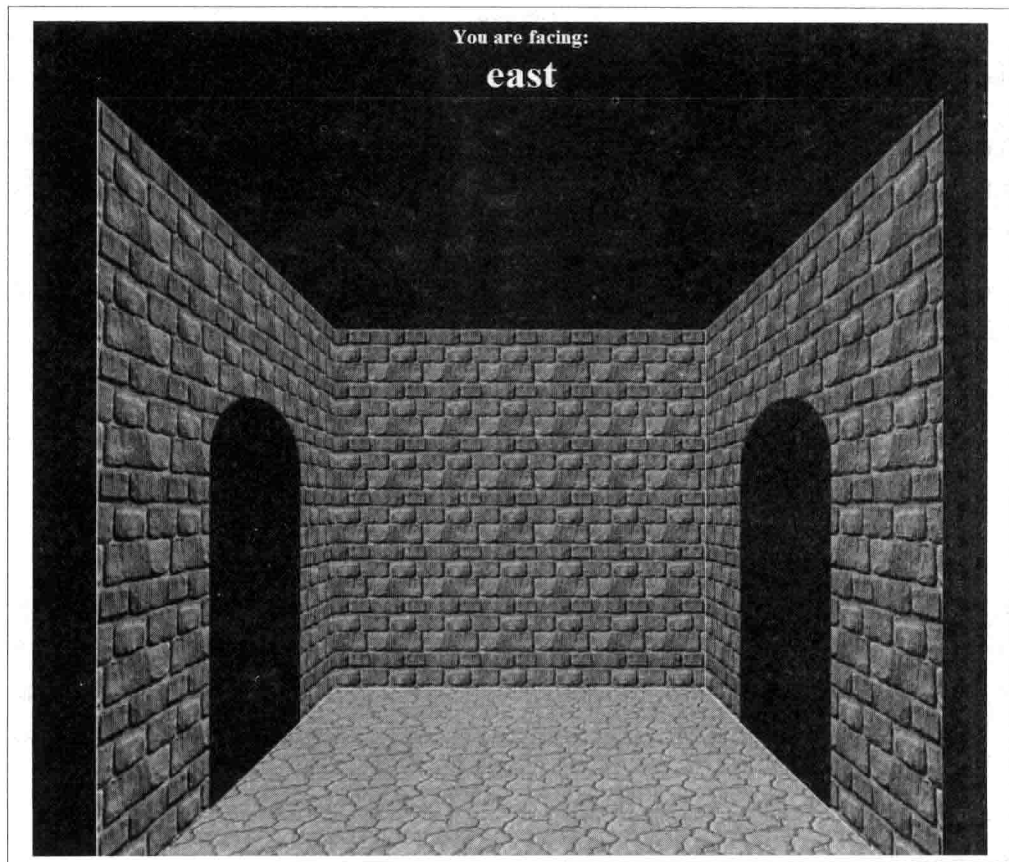


图 5-2 迷宫实例的内部

Maze+XML 格式通过模拟小鼠的视角 (rat's-eye-view) 将迷宫表示为一个由相互连接的单元格构成的网络。图 5-3 展示了如何将图 5-1 的迷宫实例表示成一个由单元格组成

的网络。我将迷宫抽象成一个网格，并为每个小方格创建一个单元格。

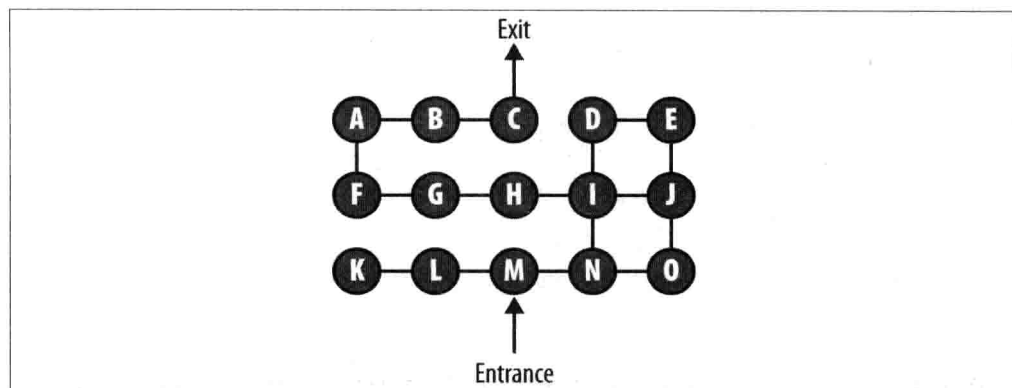


图5-3 迷宫实例——单元格的网络

Maze+XML 的单元格从 4 个方向相互连接：上北、下南、左西、右东。这就表示出口（单元格 C）在入口（单元格 M）的正北方，虽然你不能直接向北走找到出口——你需要先向东走绕个弯。

Maze+XML 迷宫里的每个单元格都是一个拥有 URL 的 HTTP 资源。如果你向这个迷宫中的第一个单元格发送一条 GET 请求，你就会收到一条表述，内容如下：

```
<maze version="1.0">
  <cell href="/cells/M" rel="current">
    <title>The Entrance Hallway</title>
    <link rel="east" href="/cells/N"/>
    <link rel="west" href="/cells/L"/>
  </cell>
</maze>
```

这条表述包含了一个人类可以理解的单元格的名字：“he Entrance Hallway”，这就和你曾经在那种很老的、文字界面的冒险类游戏里面见到的一样。除此之外，超媒体也在这里出现——这条表述还包括 `<link>` 标签，这些标签用来将这个单元格与它附近的其他单元格连接起来。从单元格 M 开始，你可以选择向西走进入单元格 L，也可以选择向东走进入单元格 N。

链接关系

上面这条表述展示了一个很强大的超媒体工具，我们将它称为链接关系。靠它们自己，`rel="east"` 和 `rel="west"` 不能表示任何含义。计算机也不会理解单词“east”和“west”的意思。但是 Maze+XML 标准为“east”以及“west”定义了明确的含义。

指的是当前资源东边的资源。当被用于 Maze+XML 媒体类型的时候，所关联的 URL 指向的是当前迷宫中东边邻近的单元格资源。

west

指的是当前资源西边的资源。当被用于 Maze+XML 媒体类型的时候，所关联的 URL 指向的是当前迷宫中西边邻近的单元格资源。

这些定义看起来好像是一种循环定义。它们仅仅是说这些链接关系 `east`、`west` 和我们平常所说的地理概念的名称是相同的。但是这些定义有助于消除语义鸿沟，这是因为，还是那句话，链接关系靠它们自己并不能表示什么含义。如果没有一个正式的定义，`east` 可以表示的是南方，而 `west` 也可以表示的是地下。

Maze+XML 标准也同样定义了链接关系：`north` 和 `south`。通过这些定义，我们可以预料到这些方向链接将会在某些 Maze+XML 表述中出现，我们也可以编写程序让计算机在遇到这样标记的时候能够理解它们：

```
<link rel="east" href="/cells/N"/>
```

在 Maze+XML 中，访问一个标有链接关系为 `east` 的链接就会使得你的客户端穿过某些抽象的地理空间移向东边。你之后就处在迷宫的另一个单元格中了。从你当前资源的所在位置中移动到东边的资源这一行为类似于你在现实生活中移动到东边，或者至少类似于你在现实生活中将手指在地图上滑动到东边。Maze+XML 就是这样解决语义挑战的：通过定义链接关系来传递应用语义。

链接关系是一个和超媒体控件（比如 Maze+XML 的 `<link>` 标签）相关的很充满魔力的字符串。链接关系说明了应用状态（对安全的请求而言）以及资源状态（对非安全的请求而言）的变化。这种变化是在客户端触发控件的时候发生的。链接关系是在 RFC5988 中正式定义的，但是这种思想其实已经存在很长时间了，几乎所有的超媒体格式都支持它们。

对于一个 RESTful API 的开发人员来说，其中一个最重要的网页就是由互联网编号分配机构（Internet Assigned Numbers Authority——IANA）（<http://www.iana.org/assignments/link-relations>）管理的链接关系注册表。我会在后面章节中对其进行讲解。它包含了 60 种已经被认为具有通用性的链接关系，并且这些链接关系不是针对某种特定的数据格式的。最简单的例子就是用于访问列表的 `next` 和 `previous` 关系。Maze+XML 规范中的 `east` 和 `west` 并不在这 60 个链接关系中；它们被认为通用性还不够高。

RFC 5988 定义了两类链接关系类型：注册关系类型（registered relation types）和扩展关系类型（extension relation types）。注册链接关系看起来很像你在 IANA 注册表中见到的链接关系：类似于 `east` 和 `previous` 这样的简短的字符串。为了避免冲突，这些短字符串需要在某些地方进行登记——不一定是 IANA，也可以是在某些标准中，比如媒体类型的定义文档中。

扩展关系看起来像 URL。如果你拥有一个域名 `mydoma.in`，你可以将一个链接关系命名为 `http://mydoma.in/whatever`，然后将它定义为你任何想要表示的内容。没有人能够定义一个和你的关系相冲突的链接关系，因为这个域名是由你来管理的。当你的用户在他们的浏览器上访问 `http://mydoma.in/whatever` 的时候，他们应该会看到这个链接关系的人类可读说明^{注 1}。

第 9 章包含一个指导来说明什么时候可以使用这种缩写的注册关系。下面是摘要：

- 任何地方你都可以使用扩展关系。
- 任何时候你都可以使用在 IANA 注册的链接关系。
- 如果一个文档的媒体类型定义了一些注册关系，你可以在这个文档中使用它们。
- 如果一个文档包含了一个定义了一些链接关系的 profile（见第 8 章），你可以在这个文档中将这些链接关系作为注册关系进行使用。
- 不要让你的链接关系的名称和 IANA 注册表中的名称发生冲突。

访问链接来改变应用状态

客户端可以通过访问一个恰当的链接（也就是：向标记为 `rel="east"` 的 URL 发送一个 GET 请求）从单元格 M“向东走”。客户端之后就会收到第二个 Maze+XML 格式的表述，内容如下：

```
<maze version="1.0">
  <cell href="/cells/N">
    <title>Foyer of Horrors</title>
    <link rel="north" href="/cells/I"/>
    <link rel="west" href="/cells/M"/>
    <link rel="east" href="/cells/O"/>
  </cell>
</maze>
```

这是地图里面的单元格 N 的 Maze+XML 格式的表述。它链接回单元格 M（使用链接关

注 1 如果你需要将某个 IANA 注册链接关系格式化为扩展关系，你可以使用这个 URI 模板 `http://alps.io/iana/relations#{name}`。对于链接关系 `author` 的可选名称就是 `http://alps.io/iana/relations#author`。这是我们提供的一个服务，它是 ALPS 项目的一部分，并不是 IANA 官方或者授权的内容，我将在第 8 章中进行描述。

系 west)，此外它还链接到单元格 I (north) 和单元格 O(east)。

客户端的应用状态也就因此发生了变化。借用 HTML 标准中的术语，客户端刚才在“访问”单元格 M，现在它正在“访问”单元格 N。现在客户端有了 3 个新的选项，这些选项是通过单元格 N 的表述中的 3 个链接来表示的。

通过访问右边的链接（北、西、西、西、北、东、东，最后继续向东），客户端可以从单元格 N 到达单元格 C。单元格 C 包含了迷宫的出口，这个出口是通过一个链接关系为 exit 的 <link> 标签来表明：

```
<maze version="1.0">
  <cell href="/cells/C">
    <title>The End of the Tunnel</title>
    <link rel="west" href="/cells/B"/>
    <link rel="exit" href="/success.txt"/>
  </cell>
</maze>
```

下面是 Maze+XML 标准对 exit 的说明：

exit

指的是表示客户端的当前所在活动或流程的出口或者终点的资源。当被用于 Maze+XML 媒体类型时，所关联的 URL 指向当前迷宫的最终出口资源。

不同于 east 和其他的方向关系，Maze+XML 并没有提供指导来说明在 exit 链接的另一端应该出现什么事物。它是一个“资源”，也就意味着，它完全可以是什么事物。在本例中，我选择将其链接到一个本文类型的祝贺消息 (success.txt)。

迷宫集合

单元格 C 通往迷宫的外部，因为它的表述中包含一个 rel="exit" 的特殊链接。但是单元格 M 作为迷宫的入口，却没有包含任何将它与其他 14 个单元格区分开来的内容。表述中也不存在 rel="entrance" 或者其他的内容。单元格 M 的标题是“The Entrance Hallway”，但是这个词组对计算机而言没有任何意义。我们如何才能消除这个语义鸿沟呢？客户端应该如何知道迷宫的起点在哪里呢？

Maze+XML 标准是通过一个集合：迷宫列表来解决这个问题的。如果你向迷宫 API 的根 URL 发送一个 GET 请求，你会收到如下 Maze+XML 格式的表述：


```

<maze version="1.0">
  <collection>
    <link rel="maze" title="A Beginner's Maze" href="/beginner">
    <link rel="maze" title="For Experts Only" href="/expert-maze/start">
  </collection>
</maze>

```

66 Maze+XML 文档中的一个集合就是一个 `<collection>` 标签，这个标签包含了一些链接关系为 `maze` 的链接。这个关系（在 Maze+XML 规范中定义，就像 `east` 和 `exit` 一样）会告诉客户端，这个链接的另一端的资源是一个迷宫的起始单元格。这个表述链接到两个迷宫：初学者迷宫（我用图 5-1 表示）以及一个更加复杂的迷宫（我现在暂时不会展示出来）。

向链接关系标记为 `maze` 的 URL（我们暂时使用 `/beginner`）发送一条 GET 请求，你就会收到一条如下内容的表述：

```

<maze version="1.0">
  <item>
    <title>A Beginner's Maze</title>
    <link rel="start" href="/cells/C"/>
  </item>
</maze>

```

这是从外部查看这个迷宫得到的一个高层次的表述。它得到了一个链接关系为 `start` 并且指向单元格 C 的链接。

Maze+XML 文档就是以此来表示 `/cells/C` 是迷宫入口这一事实。它在处在迷宫的外部视图里。一旦你进入了迷宫，单元格 C 也就没有任何特殊之处了。

指向迷宫集合的 URL 也就是那个众所周知的“广告牌上推广的 URL”。只从这个 URL 开始，所有用 Maze+XML API 能完成的事情，你都可以做了：

1. 首先获取迷宫集合的表述。因为你已经阅读过 Maze+XML 规范并将这些知识编程写进了你的客户端，所以你知道如何解析这个表述。
2. 你的客户端还知道链接关系 `maze` 表示一个迷宫。这样客户端就得到了一个能在第二次 GET 请求中使用的 URL。通过发送这个 GET 请求，你可以得到一个迷宫的表述。
3. 你的客户端知道如何解析一个迷宫的表述（因为你已经将这些知识编码到客户端里面了），然后它就知道了链接关系 `start` 表示这个迷宫的入口。这样你就可以发起第三次 GET 请求来进入迷宫了。
4. 你的客户端知道如何解析一个迷宫单元格的表述。它知道 `east`、`west`、`north`

和 `south` 的含义。所以，它可以将在抽象的迷宫里的移动过程转换为一系列的 HTTP GET 请求。

5. 你的客户端知道 `exit` 的含义，所以它知道什么时候走出了迷宫。

Maze+XML 标准还有很多其他的内容，但是你现在已经了解了它的基本内容。一个集合链接到一个迷宫，一个迷宫链接到一个单元格。你可以从一个单元格访问链接到另一个单元格的链接。最终，你会发现一个通向迷宫外部的标记为 `exit` 的链接的单元格。这些信息已经足够开始编写客户端了。

67

Maze+XML是API吗？

如果你现在已经从这个领域中得到一些经验，你可能想要知道：API 在哪里？迷宫游戏不是一个复杂的应用程序，但是虽然如此，你所希望的可能还不仅仅是有一些 XML 标签名以及链接关系。Maze+XML 规范并没有那些你习以为常的东西。规范没有定义任何的 API 调用或者提供任何关于构造 URL 的规则。事实上，它几乎没有提到过 HTTP。我已经在例子表述中展示了一些 URL，但是我故意没有保持 URL 格式设计的内部一致性（比较 `/beginner` 和 `/expert-maze/start`），所以你就不会认为 URL 格式是由 Maze+XML 标准定义的了。

你已经习惯了的东西是很危险的。在供某个组织使用的应用程序中，基于 API 调用的设计会效果很好并且易于开发。API 调用的说法假设打破了网络边界，并且允许客户端像调用本地代码库的 API 一样来调用远程计算机的方法。已经有很多的书和软件工具来帮助你进行那样的设计。

我的经验告诉我“API 调用”的说法不可避免地会将服务器的实施细节暴露给客户端。这就引入了服务器代码与客户端代码的耦合。当涉及这些 API 的人们都是自己的朋友和同事的时候，影响还不大。

但是本书关注的是 web API，也就是说，web 规模的 API（任何公共成员都可以在 web 上使用客户端、编写客户端，或者某些情况下，还可以编写服务器）。当你允许你所在组织之外的某个人发起 API 调用的时候，你也使得这个人成为了你的服务器实现中的一个无声的搭档。这也就使得在不伤害这个未知客户的情况下对服务器端进行任何改动都变得异常困难。

这就是公共 API 的改动相当罕见的原因。你无法在不给用户造成巨大痛苦的情况下改变基于 API 调用的 API，你仅仅能做到的是改变本地代码库的 API 而不造成痛苦。在 web 级的规模上，API 调用的设计方式是无能为力的。

基于超媒体的设计拥有更多的灵活性。每次客户端发送一个 HTTP 请求，服务器就会发送响应来说明哪些 HTTP 请求作为下一步的选择是最合理的。如果服务器端的选项发生了变化，那么那个响应文档也就会相应发生变化。这不能解决我们所有的 API 问题——语义鸿沟还是一个很大的问题！——但是它至少解决了那些我们已经知道如何解决的问题。

客户端1：游戏

Maze+XML API 的一个很明显的用途就是开发供人类玩乐的游戏。这里有一个单页应用程序，这个程序可以获取一个迷宫集合供你从中选择一个迷宫来玩。一旦你进入了一个迷宫，你所看到视图就像一只老鼠看到的视图一样，你可以通过输入方向来游览这个迷宫；一旦你找到了出口，你就得到一个分数——你在迷宫中移动的次数^{注2}。

我们倾向于将“API 客户端”理解为一个自动化的客户端。但是像上面例子一样的由人类驱动的客户端在现代的 API 生态系统里也起了很大的作用。这在移动应用中很常见，这些移动应用就是由人类来驱动进而通过 web API 和服务器进行通信的。最好的办法就是，将人类添加到环节里面，这样前面所讨论的语义鸿沟也就不是问题了。

图 5-4 展示了游戏客户端加载进浏览器以后的界面。

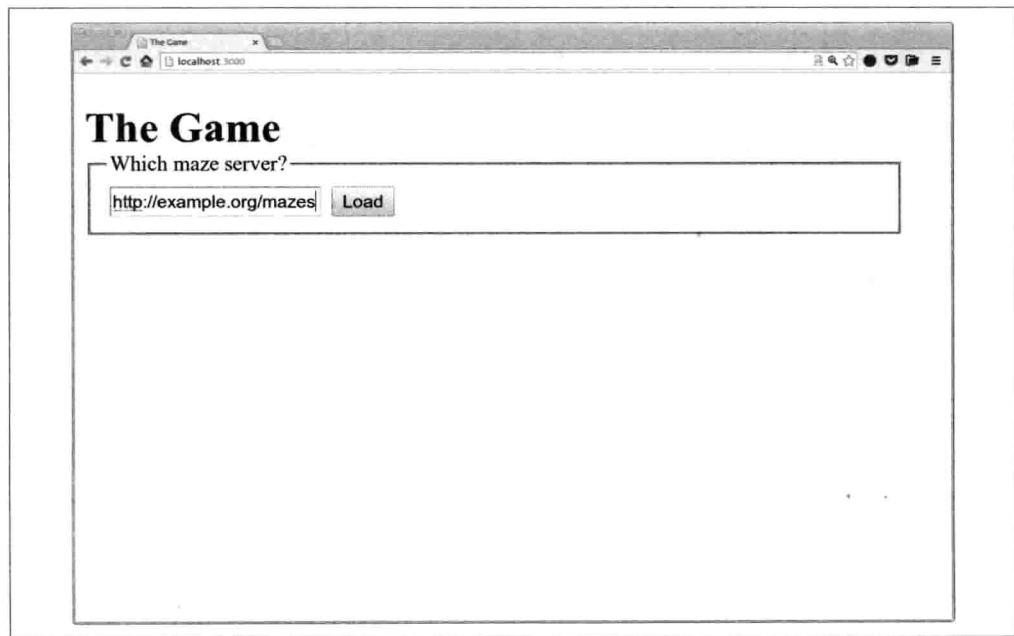


图5-4 游戏客户端的初始状态

注2 你可以在 *RESTful Web APIs* 的 Github 仓库中找到游戏客户端的 Node 源码(查看 *Maze/the-game/* 目录)。

我输入一个广告牌 URL——也就是那个迷宫集合的 URL——单击 Load 按钮。游戏客户端就会向我输入的那个 URL 发起一次 HTTP GET 请求：

```
GET /mazes/ HTTP/1.1
Host: example.org
Accept: application/vnd.amundsen.maze+xml
```

服务器以 Maze+XML 格式的文档做出响应：

```
<maze version="1.0">
  <collection href="http://example.org/mazes/">
    <link href="http://example.org/mazes/a-beginner-maze" rel="maze"
      title="A Beginner's Maze" />
    <link href="http://example.org/mazes/for-experts-only" rel="maze"
      title="For Experts Only" />
  </collection>
</maze>
```

游戏客户端读取这个文档——迷宫集合的表述，然后将其翻译到 HTML 界面（见图 5-5）。我就会看到两个迷宫选项。它们分别对应 Maze+XML 文档中链接关系为“maze”的两个链接。



图5-5 迷宫选项

我在文本框中输入“1”来选择一个迷宫，然后单击 Go 按钮，这样我就进入了初学者迷宫的内部了。图 5-6 展示了客户端是如何呈现迷宫的第一个宫格的。

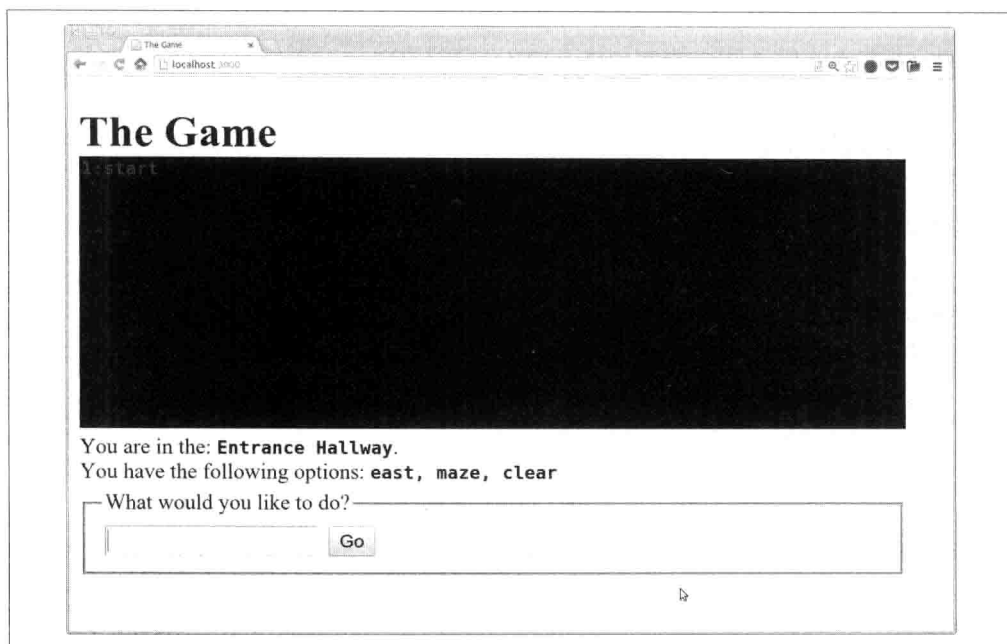


图5-6 初学者迷宫的第一个单元格

这是如何发生的呢？是通过超媒体。当我输入“1”的时候，我就是告诉客户端让其发起一个 HTTP GET 请求来访问那第一个 `rel="maze"` 的链接。这个接口与单击链接的接口有很大的不同，但是它们的效果是相同的。

请求内容如下：

```
GET /mazes/beginner HTTP/1.1
Host: example.org
Accept: application/vnd.amundsen.maze+xml
```

下面是服务器发送的 Maze+XML 格式的文档作为响应：

```
<maze version="1.0">
  <item href="http://example.org/maze/beginner" title="A Beginner's Maze">
    <link href="http://example.org/mazes/beginner/0" rel="start"/>
  </item>
</maze>
```

由于在这个文档中只有一个链接：指向迷宫起点的链接，人类没有其他选择可选，所以不需要做出决策。游戏客户端甚至不会将这个表述显示出来，取而代之的是，客户端会按照预先编写好的程序的设定而自动访问那个 `rel="start"` 的链接。这就表示发起一个新的 GET 请求：

```
GET /mazes/beginner/0 HTTP/1.1
Host: example.org
Accept: application/vnd.amundsen.maze+xml
```

这个 GET 请求会收到这个迷宫之中的一个单元格的表述：

```
<maze version="1.0">
  <cell href="http://example.org/mazes/beginner/0" rel="current"
    title="Entrance Hallway">
    <link href="http://example.org/mazes/beginner/5" rel="east"/>
    <link href="http://example.org/mazes/beginner/J" rel="south"/>
  </cell>
</maze>
```

这些信息在被翻译成 HTML 以后就会被展示给人类用户。这也就是在我最终看到图 5-6 的界面之前所发生的一切。

现在，我就已经处在“初学者迷宫”的内部了。从此以后，我就可以参观这个迷宫了，我会从所提供的列表选择一个方向（east、north 等）将其输入文本框。每次我单击一下 Go 按钮，我都是在告诉客户端去发起一次 HTTP GET 请求来访问所对应的链接。图 5-7 展示了我在穿过初学者迷宫的过程中位于单元格 G(“The Tool Room”)时的情况。

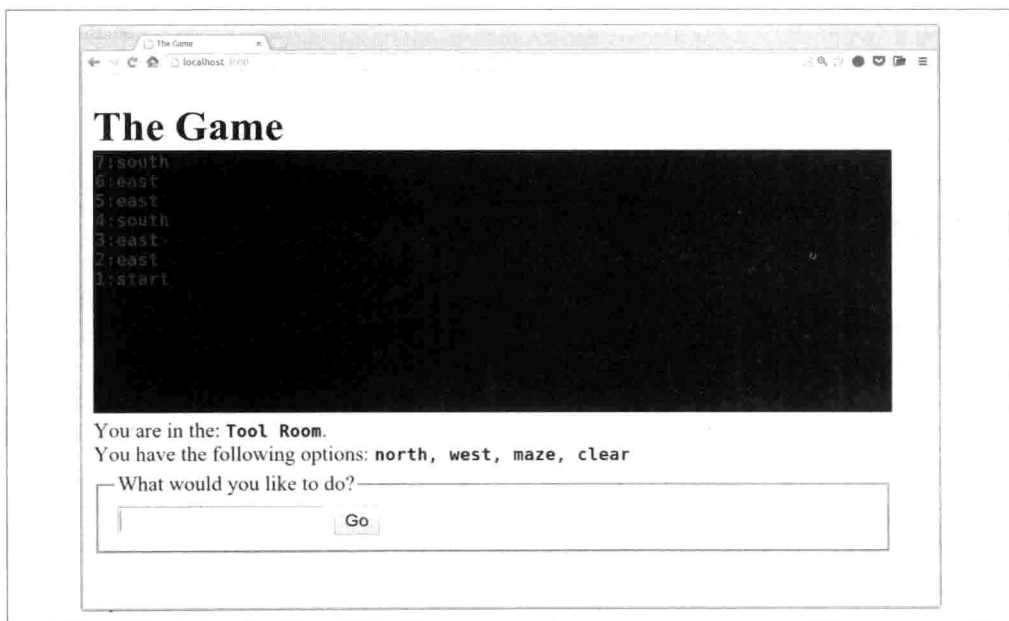


图5-7 初学者迷宫的中间

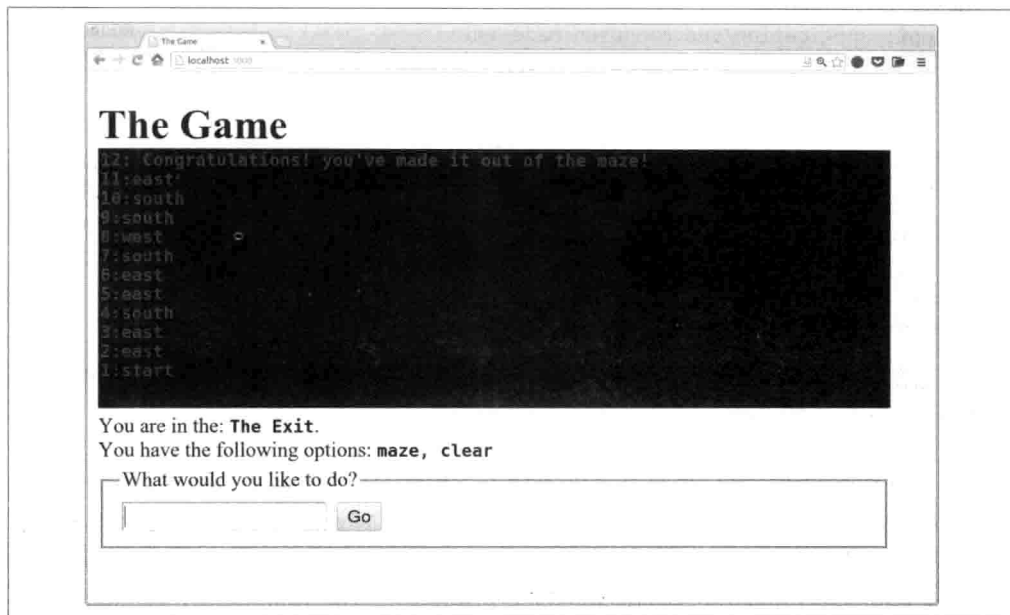


图5-8 初学者迷宫的外部

当我离开迷宫时，客户端展示了一条位于 *exit* 链接另一端的祝贺消息（资源状态的一部分）。

Maze+XML服务器

我还会编写另外两个 Maze+XML 客户端，但是在此之前，我要先介绍一下服务器端的实现。本章的所有客户端都是运行在一个非常简单的服务器上的，这个服务器是我按照 Maze+XML 标准专门为本书开发实现的^{注3}。

当今大部分的 API 都是 fiat 标准，这些 API 以某家特定的公司为后盾，并且只存在于一台主机上，但是 Maze+XML 是一个个人标准，任何人都可以对其进行开发实现。这就意味着可以有无数的 Maze+XML 服务器，无数的服务器实现。我的服务器实现也并没有什么特别之处。事实上，它的能力相当有限。它只能提供 Maze+XML 迷宫的一个很小的子集：整洁的、功能良好的、能够适应基于 JSON 的文件格式的迷宫。

注3 服务器端代码在 *RESTful Web APIs* 的 Github 仓库（查看 *Maze/server*）目录中。

我的服务器并不意味着就是最好的 Maze+XML 实现，但是易于向它添加新的迷宫进去。我的服务器是用简单的 JSON 文档来保存迷宫数据的。下面的 JSON 文档所代表的迷宫就是我之前用来作为例子的那个“初学者”迷宫。这并不是 REST 意义上的迷宫的表述，因为它永远不会通过 HTTP 被发送出去。它只是原始数据，这些原始数据被用于生成能通过 HTTP 发送出去的 Maze+XML 文档：

```
{
  "_id" : "five-by-five",
  "title" : "A Beginner's Maze",
  "cells" : {
    "cell0":{"title":"Entrance Hallway", "doors":[1,1,1,0]},
    "cell1":{"title":"Hall of Knives", "doors":[1,1,1,0]},
    "cell2":{"title":"Library", "doors":[1,1,0,0]},
    "cell3":{"title":"Trophy Room", "doors":[0,1,0,1]},
    "cell4":{"title":"Pantry", "doors":[0,1,1,0]},
    "cell5":{"title":"Kitchen", "doors":[1,0,1,0]},
    "cell6":{"title":"Cloak Room", "doors":[1,0,0,1]},
    "cell7":{"title":"Master Bedroom", "doors":[0,0,1,0]},
    "cell8":{"title":"Fruit Closet", "doors":[1,1,0,0]},
    "cell9":{"title":"Den of Forks", "doors":[0,0,1,1]},
    "cell10":{"title":"Nursery", "doors":[1,0,0,1]},
    "cell11":{"title":"Laundry Room", "doors":[0,1,1,0]},
    "cell12":{"title":"Smoking Room", "doors":[1,0,1,1]},
    "cell13":{"title":"Dining Room", "doors":[1,0,0,1]},
    "cell14":{"title":"Sitting Room", "doors":[0,1,1,0]},
    "cell15":{"title":"Standing Room", "doors":[1,1,1,0]},
    "cell16":{"title":"Hobby Room", "doors":[1,0,1,0]},
    "cell17":{"title":"Observatory", "doors":[1,1,0,0]},
    "cell18":{"title":"Hot House", "doors":[0,1,0,1]},
    "cell19":{"title":"Guest Room", "doors":[0,0,1,0]},
    "cell20":{"title":"Servant's Quarters", "doors":[1,0,0,1]},
    "cell21":{"title":"Garage", "doors":[0,0,0,1]},
    "cell22":{"title":"Tool Room", "doors":[0,0,1,1]},
    "cell23":{"title":"Banquet Hall", "doors":[1,1,0,1]},
    "cell24":{"title":"Spoon Storage", "doors":[0,0,1,1]}
  }
}
```

每个单元格都是由一个名称 ("title") 以及一个由二进制数字组成的列表 ("doors") 来表示的，其中这个数字列表表示的是在北、西、南、东四个方向上是否有门。这些罗列出来的大小相同的单元格组成了一个 5×5 的二维网格。像这样的迷宫被称为完美的迷宫 (perfect mazes)^{注4}，人们很容易找到它们的出口从而走出迷宫。我的服务器所能理解的迷宫也就如此而已。但是 Maze+XML 媒体类型可以展示各种不同大小的二维迷宫拓扑结构——想象一下单行道的迷宫或者随机生成的无限大的迷宫。

注4 你可以在 Astrolog 的迷宫分级页面 (<http://www.astrolog.org/labyrnth/algrithm.htm>) 找到更多这一主题的信息。

客户端2：地图生成器

游戏客户端依靠人类来决定往哪里走。但是现实中已经有很多用于自动穿越迷宫的算法，我们没有理由写不出一个能跟这个手动操作的客户端相匹配的自动化的客户端。

鉴于我已经编写过一个目标是穿越迷宫的客户端（客户端1：游戏）了，为了保持内容的趣味性，这个新的客户端会做一些稍微不同的事情。我将它称为地图生成器，它是一个用于绘制迷宫地图的客户端（地图生成器的代码在 *RESTful Web APIs* GitHub 仓库中的 *Maze/the-mapmaker* 目录下）。这个客户端会试图访问迷宫里的每一个单元格并且构造出一张能够直观地展示出来的地图。这个客户端不会试图离开这个迷宫，它希望能看见全部的东西。当它发现出口以后，它会将出口标记在地图的内部表述里，然后继续移动。它不会访问“exit”链接。

游戏客户端是一款用 Node 编写的运行在 web 浏览器中的 web 应用程序。这个地图生成器也是用 Node 编写的，但是它是一款命令行应用程序，它会将输出（output）打印到控制台中。如果你为这个客户端提供整个 Maze+XML 系统的广告牌 URL 的话，这个程序会绘制出网站上所有的迷宫的地图。如果你给客户端提供的是某一个迷宫的 URL 的话，它将只绘制出这个迷宫的地图。当我将地图生成器客户端运行在初学者迷宫上时，地图生成器程序会输出如下的 ASCII 码样式的结果。

```
$ node the-mapmaker http://localhost:1337/mazes/tiny
Exploring A Beginner's Maze...
```

```
+---+---+---+---+
|   |   |   |   |   |
| s 00 | 05 | 10 | 15 | 20 |
|   |   |   |   |   |
+---+---+---+---+
|   |   |   |   |   |
| 01 | 06 | 11 | 16 | 21 |
|   |   |   |   |   |
+---+---+---+---+
|   |   |   |   |   |
| 02 | 07 | 12 | 17 | 22 |
|   |   |   |   |   |
+---+---+---+---+
|   |   |   |   |   |
| 03 | 08 | 13 | 18 | 23 |
|   |   |   |   |   |
+---+---+---+---+
|   |   |   |   |   |
| 04 | 09 | 14 | 19 | 24 E
|   |   |   |   |   |
+---+---+---+---+
```

Map Key:
S = Start
E = Exit
0:Entrance Hallway
1:Hall of Knives
2:Library
3:Trophy Room
4:Pantry
5:Kitchen
6:Cloak Room
7:Master Bedroom
8:Fruit Closet
9:Den of Forks
10:Nursery
11:Laundry Room
12:Smoking Room
13:Dining Room
14:Sitting Room
15:Standing Room
16:Hobby Room
17:Observatory
18:Hot House
19:Guest Room
20:Servant's Quarters
21:Garage
22:Tool Room
23:Banquet Hall
24:Spoon Storage

服务器并没有用这种图形化的格式定义迷宫；这些迷宫都是采用 JSON 文档保存的并通过 XML 文档对外提供服务。地图生成器程序通过自动探测来构造出了迷宫的这幅图形化视图。

当地图生成器进入一个迷宫时，它会识别出第一个宫格中的所有门口（链接），然后每次访问其中的一个链接，它会非常高效地从一个宫格瞬间移动到另一个宫格而不用麻烦地原路返回。在每个宫格里面，地图生成器会查看所有的出口并构造出一个它还需要继续访问的宫格的列表。地图生成器会很有效率地对迷宫执行一次广度优先搜索。

一旦所有的单元格（以及单元格之间的所有链接）都被解析过，地图生成器就会利用它所收集来的数据来生成一个 ASCII 码地图，这个地图可以展示出这些单元格以及它们之间的关系。

和大多数的 API 客户端相比，地图生成器客户端拥有更广的应用状态视图。游戏客户端会像人类穿过迷宫那样来行动。你总是正在“访问”某一个特定的单元格，并且你只能移动到和你当前所访问单元格直接相邻的单元格。这些相邻的单元格就是你可能的下一

个状态。当你输入一个方向时，你就从这些可能的状态中选择一个，而放弃了其他的状态。你在不断地前进。Web 浏览器就是这样工作的。

76 地图生成器不会一直前进。就它而言，它所看到的每个链接都是一个可能的下一个状态。它不会像人一样穿过迷宫，它会像菌类一样扩散，直到它占领了迷宫的所有单元格。

从服务器的角度看，地图生成器好像是发疯似地在迷宫里面来回移动。这很少见，但是就 Maze+XML 规范而言，这是完全合法的。地图生成器仅仅是保持的应用状态比游戏客户端要多一些而已。

客户端3：吹牛者

Maze+XML 标准定义了用 XML 文档展示迷宫以及迷宫集合的方式。它并没有说明这些迷宫是做什么用的。面对一个迷宫，人类的本能倾向是寻找出口走出迷宫。游戏客户端就重现了这样的经验。但是 Maze+XML 并没有要求客户端用人类的方式来穿过迷宫。

我们已经看到了：地图生成器客户端一直在迷宫里来回地瞬间移动，它从来不会访问“exit”链接。它会在里面四处跳跃直到整个迷宫都被绘制出来，然后它就会真正停止发起 HTTP 请求。这看起来和迷宫的用途相反，但是谁会这么说呢？

我的第三个 Maze+XML 客户端，吹牛者，将这个逻辑发挥到了极致。这个客户端甚至从没进入过迷宫。它读取迷宫集合，随机选取一个，然后很绝对地声称已经走出了迷宫^{注5}。过程如下：

```
$ node the-boaster http://example.org/mazes
Starting the maze called: For Experts Only...
*** DONE and it only took 2 moves! ***
```

很明显，你不可能在两步之内从这个专家（Experts）迷宫中走出来。这个吹牛者客户端甚至都没有试过。它首先向 <http://example.org/mazes> 发起一次 HTTP 请求，接着读取迷宫集合，然后选择“For Experts Only”，最后声称已经在不切实际的步数内走出了迷宫。

这是欺骗吗？在穿越迷宫问题的范围内，这当然属于欺骗。但是在 REST 或者与 Maze+XML 标准兼容的世界里，这完全是正当的。这个吹牛者客户端的确知道 `vnd.amundsen.application/maze+xml` 文档的含义，它也知道那些带有 `rel="maze"` 的链接指向迷宫，它只是懒得去穿过这个迷宫。

注5 Boaster 的代码在 *RESTful Web APIs* 的 Github 仓库中（查看 *Maze/the-boaster* 目录）。

客户端做自己想要做的事

这里有 3 个客户端：游戏客户端、地图生成器客户端、吹牛者客户端。它们都是首先从理解 Maze+XML 媒体类型来开始工作的，但是它们的目标不同，所以它们用相同的数据做着不同的事情。

这没有问题。服务器的职责是以一种客户端可以参与其中的方式描述迷宫，而不是给客户端下达目标命令。Maze+XML 规范描述的是一个问题空间，而不是客户端和服务端之间指定的关系。客户端和服务端必须对传递于它们之间的表述达成共识，但是它们不需要在要解决什么问题上有相同的看法。

对标准进行扩展

Maze+XML 是一个无聊的问题领域中的一个特别设计的例子。但是让我们想象一下这样的场景：某个人确实想要提供超媒体迷宫服务，或者是作为某项业务的一部分，抑或是仅仅为了娱乐。这些需求都不会自动使得 Maze+XML 成为正确的选择。甚至于当已经存在有某个面向你的问题领域的标准的时候，它还是有可能无法完全满足你的需求。

任何真正想要采用 Maze+XML 的人都将不会满足于标准中提到的内容。这个标准将你限定在使用东南西北 4 个基本方向的二维迷宫中。这不是很好玩。如果我想要提供三维迷宫的服务，我该怎么办呢？

仅仅为了支持三维迷宫，而从零开始创造一个崭新的标准是很愚蠢的行为。Maze+XML 标准已经几近足够好了。我只需要对其进行稍微的扩展使得它可以支持两个新的方向：up 和 down 就足够了。

幸运的是，Maze+XML 明确地允许这种类型的扩展（见 Maze+XML 规范第 5 节）。只要我不重新定义一些规范中已有的内容，我可以向 Maze+XML 文档中添加任何我想要的内容。为了得到我所想要的三维迷宫，我将仅仅在这里定义两个新的链接关系就足够了：

up

指向位于当前资源的上方的资源。

down

指向位于当前资源的下方的资源。

这是一个简单的扩展，但是它完全改变了迷宫的样式以及迷宫在服务器中的保存方式。我的服务器实现是用一个以单元格为元素填充的二维数组来保存迷宫的，每个单元格周围可能有4个单元格。为了支持这两个新的关系，我需要修改服务器代码来反映如下事实：一个迷宫是一个三维数组，每个单元格周围可能有6个单元格。

但是客户端不会看到很大的变化。客户端看到的仅仅是在表述中增加了两个新的链接关系：

```
<maze version="1.0">
  <cell href="/cells/middle-of-ladder">
    <title>The Middle of the Ladder</title>
    <link rel="up" href="/cells/top-of-ladder"/>
    <link rel="down" href="/cells/bottom-of-ladder"/>
  </cell>
</maze>
```

这些使得 Maze+XML 标准看起来过于简单的事物成功地将所有新增加的服务器端复杂度都对客户端隐藏起来。Maze+XML 标准并没有在迷宫“应该”是什么样子的问题上说明太多。为相互连接的单元格定义两个新的连接方式需要对服务器端实现进行完整的重新设计，但是修改之后的表述仍然与 Maze+XML 标准兼容，客户端还是可以解析它们的。

但是这并不表示客户端能够自动理解这些新的应用语义。考虑一下当我们分别为游戏客户端、地图生成器、吹牛者客户端提供一个三维迷宫的时候，将会发生什么事情呢？

令人惊讶的是，游戏客户端运行正常！这个客户端的代码并没有写死只有4个基本方向。它被编程设计为将发现的每个链接都展示给用户，让用户从中选择一个链接。因为我将新的链接关系命名为“up”和“down”，所以当人类穿过三维迷宫的时候，他会看到如图 5-9 所示的界面。

这些选项对人类用户来说是有意义的，如果用户输入“up”，客户端就会访问 rel="up" 的链接。向 Maze+XML 增加新的应用语义并不需要对游戏客户端进行任何的改动，因为这个环节中有人类的参与。

吹牛者客户端在三维的迷宫中运行得也很顺利，因为它甚至都不会进入迷宫之中。事实上，不论对迷宫进行怎么样的扩展，吹牛者客户端都应该能工作于任何与 Maze+XML 相兼容的服务器上。

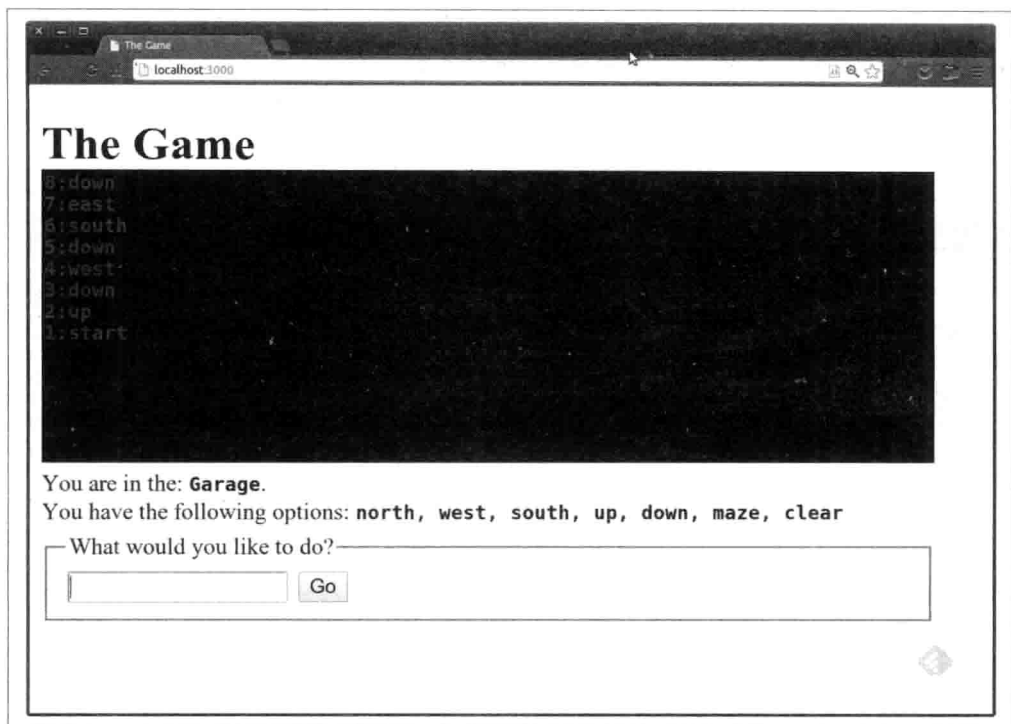


图5-9 游戏客户端自动支持“up”和“down”

但是地图生成器客户端在遇到三维迷宫的时候，就全然不得要领了。考虑如下的表述：

```
<maze version="1.0">
  <cell href="/cells/bottom-of-ladder">
    <title>The Bottom of theLadder</title>
    <link rel="up" href="/cells/middle-of-ladder"/>
    <link rel="east" href="/cells/tunnel"/>
    <link rel="north" href="/cells/underwater-garden"/>
  </cell>
</maze>
```

如果地图生成器得到如上所述的表述，它会访问“east”和“north”链接，但是绝对不会访问“up”链接。让客户端在这个三维迷宫中自由移动，它将会绘制出迷宫的某一层的地图。它所看到的只是迷宫的一个二维切片。

这是可以理解的。我们不能期望客户端能够理解那些并没有编程实现的链接关系。但是你可能也不会预料到，即便将这些新的连接关系简单地告诉给地图生成器，这也并不会有任何帮助！

即使地图生成器知道如何访问一个“up”连接，但是它依旧不知道如何展示它在该链接的另一端发现的资源。和我们的服务器实现的例子一样，地图生成器客户端也只有一个二维的思维。它生成的是一个二维的 ASCII 码地图。一个三维的迷宫完全不能和地图生成器客户端相兼容。

地图生成器的缺陷

事实上，即便是运行在一个没有使用任何 Maze+XML 扩展的迷宫，地图生成器也同样可能会失败。如果服务器端发送如下表述，地图生成器会如何处理呢？

```
<maze version="1.0">
  <cell href="/cells/44">
    <title>Hall of Mirrors</title>
    <link rel="east" href="/cells/45/>
  </cell>
</maze>
```

如果服务器接着发送 east 链接所对应的另一端的表述呢？

```
<maze version="1.0">
  <cell href="/cells/45">
    <title>Mirrored Hall</title>
    <link rel="west" href="/cells/129/>
    <link rel="east" href="/cells/44/>
  </cell>
</maze>
```

它们两个都是合法的 Maze+XML 文档。但是它们所描述的迷宫确是一个非欧几里得地图。从“Hall of Mirrors”的单元格向东走进入“Mirrored Hall”单元格，然后继续向东走，你却回到了“Hall of Mirrors”单元格。你可能在很多电子游戏中见到过这种恶作剧的把戏。这是一个完全合法的没有使用任何 Maze+XML 扩展的迷宫，但是地图生成器会在试图绘制这个迷宫的地图的时候崩溃掉。

看起来好像是地图生成器在设计之初就有一个隐藏的前提假设！那就是：服务器端只提供整齐的、能够用网格表示的迷宫。而我们的服务器实例也只提供这种类型的迷宫，这并非巧合。我在设计地图生成器客户端的时候就考虑了特定的服务器。这也就证明我们的这个客户端并不能工作于符合 Maze+XML 规范的全部迷宫。它只能工作在那些你可以在我们的服务器上能找到的迷宫。

我发现这个规律具有普遍适用性。某个为特定的服务器实现编写的客户端可以针对该服务器上的某些古怪的问题进行特别优化，但是如果你试图让它运行在相同标准的另一种服务器实现上，它很快就会失败。这并不意味着地图生成器是一个完全没用的客户端，

它只不过是只能给某些迷宫绘制地图而已。

想象一下这样的场景，你要启动一个只在某个特定的网站上测试过的 web 浏览器。一旦你将该浏览器访问一个没有被测试过的网站，浏览器很可能就会崩溃。情况就是这样，类似于 Maze+XML 的标准可能有很多服务器端实现，客户端实现需要被设计成能工作于所有的服务器实现，而不是仅仅某个服务器实现。

修复（以及修复后的瑕疵）

我们可以修复这个地图生成器吗？一个“修复方案”就是让客户端去检测每个新发现的单元格是否符合它所构造的网格结构。和程序崩溃不同的是，当它检测出在同一个网格区域中有两个不同的单元格时，它就会打印一条错误消息然后优雅地退出。

但是那样做仅仅是避免了崩溃而已。我们已经给客户端提供了正好足够的情报用来识别出它不能理解的迷宫。如果我们希望客户端能够真正理解那些并不完美的迷宫，那么这个“网格”的数据结构就不能再用了。我们应该使用的正确的数据结构是有向图（directed graph）。

我们可以编写一个更好的地图生成器，这个地图生成器在迷宫中穿梭的时候会构造出一张有向图，然后使用像 force-directed graph drawing 这样的算法来将有向图呈现出来。对于一个或多或少符合网格结构的迷宫，改进后的地图生成器会将将有向图渲染为和之前的老式地图生成器的 ASCII 码图看起来很相似的图形（见图 5-10）。

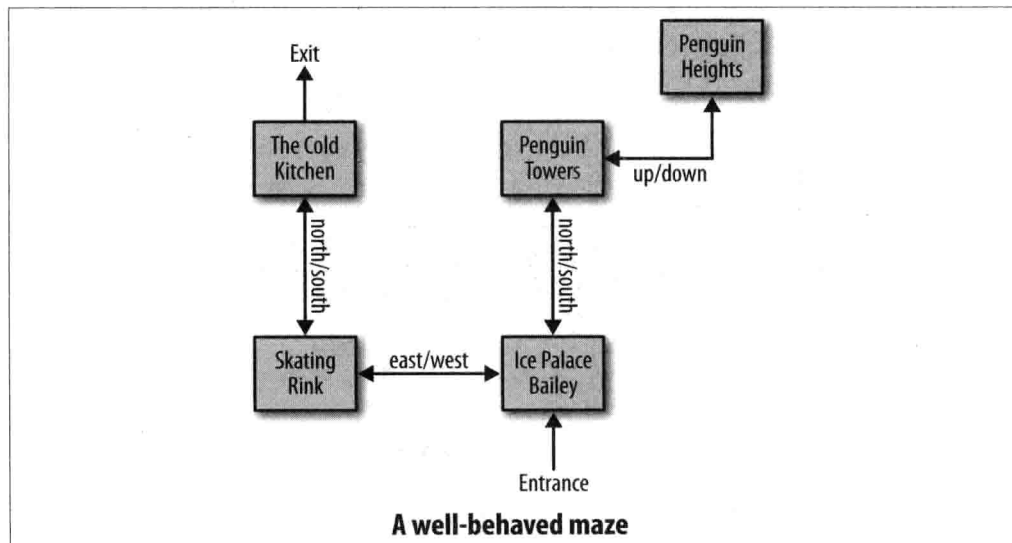


图5-10 被看作有向图的功能良好的迷宫

图 5-10 就是地图生成器所生成的地图，它将书页的上部定义为“north”（北）。对于如何处理“up”和“down”，还不确定，但是可以断定的是，它们很可能是相反的方向，并且将它们以直观的方式展示出来是可行的。

现在想象一下一个拥有无数环路和单行道的恶作剧的迷宫。这样的迷宫很可能会使得那个老式的、改进前的地图生成器崩溃，但是经改进后的地图生成器可以为该迷宫呈现出一张图，如图 5-11 所示。

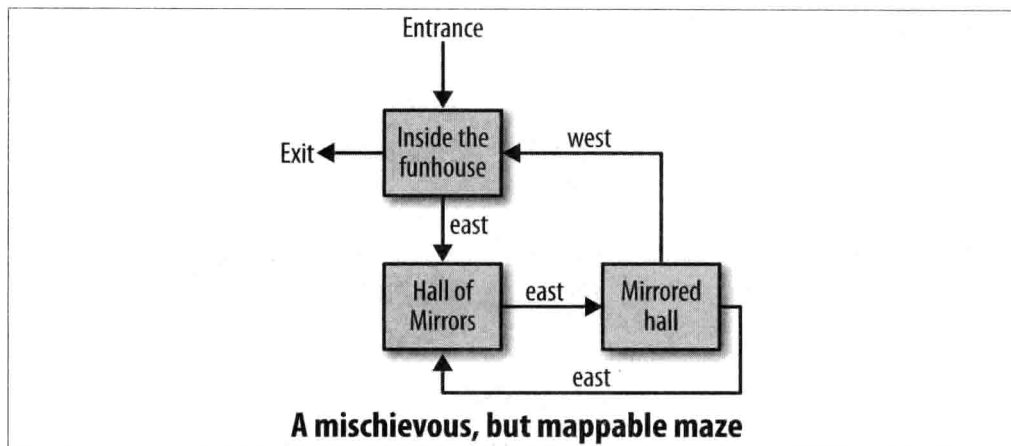


图5-11 恶作剧迷宫

现在的地图生成器已经完美了吗？不幸的是，还没有。这个改进后的地图生成器依旧包含一些隐藏的前提假设。图 5-12 展示了一个无限大的迷宫。人们很容易找到这个迷宫的出口从中出去，但是将这个迷宫的地图绘制出来却是不可能的。

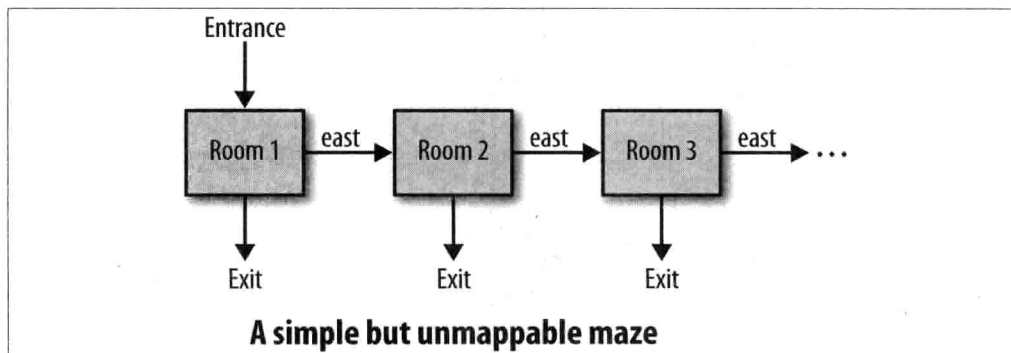


图5-12 一个简单但是无法绘制地图的迷宫

任何地图生成器客户端都不可能为一个无限大的迷宫绘制出地图来：它永远不会有机会

将地图绘制出来。这并不意味着地图生成器是没有用的。这只是说它并不能处理所有符合 Maze+XML 标准的独立迷宫。

迷宫的暗喻

再回过头来看看图 5-10 至图 5-12。将它们与图 1-9 做一下比较，图 1-9 是我之前用来展现网站结构的有向图。

它们的相似性并不是偶然的。正如我在本章开始所讲的那样，对于超媒体应用而言，迷宫通常是一个很好的暗喻。有些“迷宫”很整齐并且功能良好，而有一些迷宫就非常混乱并且无限大。将状态图看作一个需要游览的迷宫，这样你的思路就能进入理解超媒体 API 正确的轨道了。

解决语义鸿沟

对于一个领域特定 API 的设计者而言，消除语义鸿沟需要两个步骤：

1. 将你的应用语义写到一个人类可读的规范（比如 Maze+XML 标准）中。
2. 为你的设计注册一种或者多种 IANA 媒体类型（就像 `vnd.amundsen.application/maze+xml`）。在注册处，将这些媒体类型与你编写的人类可读的文档关联起来。在第 9 章，我将讨论媒体类型的命名和注册流程。

你的客户端开发人员可以颠倒这个过程从反方向来消除语义鸿沟：

1. 在 IANA 注册表中查找陌生的媒体类型。
2. 通过阅读规范来了解如何处理这些陌生的媒体类型的文档。

这里并没有捷径可走。为了使客户端代码能正常工作，你的用户将必须阅读你编写的文档进而做一些其他的工作。我们不可能完全摆脱语义鸿沟，因为计算机还不能像人类一样聪明。

领域特定设计在哪里？

当你想要发布一个 API 的时候，首先要做的是找到一个已有的领域特定设计。重复造轮子是没有意义的。

即便如此，你也不大可能找到一套完整的解决方案。领域特定的数据格式数以百计，但是它们中的很多格式并不包含超媒体控件。在第 10 章中，我会介绍一些例外的格式，比如 VoiceXML 和 SVG。你可能会采纳的领域特定设计是问题细节的文档，一种简单的基

于 JSON 的用于描述错误条件的格式（同样会在第 10 章中讨论到）。

84

但是一个数据格式不包含超媒体控件，并不意味着这个数据格式就是没有用处的。在第 8 章，我将为你展示 JSON-LD 是如何来为任意的 JSON 格式添加基本的超媒体能力的。在第 10 章，我将同样展示 XForms 和 XLink 如何来为 XML 格式完成同样的功能。这些技术可以让你将超媒体控件移植到现有的并不包含超媒体控件的 API 上。

最终的奖赏

我们在超媒体 API 中依旧可以找到那些不支持超媒体的格式的用武之地。考虑一下 JPEG 图片格式。关于该格式的文档记录很完善，并且它也拥有一个注册的媒体类型（`image/jpeg`），没有什么能比二进制的图片文件更好地用于展示相片的了。但是你不能使用 JPEG 来作为 Web API 的主要内容，除非你想要设计一个只提供 JPEG 图片的网站。我们也没有办法将一张 JPEG 图片链接到另一张图片。

用于管理相片的 web API 很可能会用 JPGE 格式来发送和接收表述。仅仅因为 JPGE 格式没有超媒体控件就构造你自己的二进制图片格式是很愚蠢的行为。但是 JPEG 格式并不是基于超媒体的相册 API 的核心内容，这项荣誉应该属于像 HTML 这样的格式。虽然 HTML 不能直接展示一张相片，但是它可以将照片嵌入一个文本文档中、将相片及它的标题配对、展示一个相片列表以及展示用于搜索和标记相片的表单。

一个 `image/jpeg` 表述将是客户端浏览相片 API 的超媒体“迷宫”并定位到一个特定的相片后得到的奖品。这个“迷宫”自己是使用支持超媒体控件的文档格式描述的。这两种格式共同工作组成了一个完整的 API。

报头中的超媒体

我在第 4 章中已经展示过如何使用 HTTP 报头 Link 来为那些没有超媒体控件的文档添加简单的超媒体链接和表单。通过使用这些报头，你可以令人信服地设计出一个只提供 JPEG 图片的 API，但是我不建议你这么做。

抄袭应用语义

这是一个完全不同的技术。我想要首先介绍一下 vCard 格式，它是由 RFC6350 定义的，并且分配有一个媒体类型 `text/vcard`。这是一个设计用来交换业务名片上的各种个人信息的领域特定的纯文本的格式。听起来很有用，是吗？许多 web API 都要处理与人和业务相关的信息。

如下是一个简单的 vCard 表述：

```
BEGIN:VCARD
VERSION:4.0
FN:Jennifer Gallegos
BDAY:19870825
END:VCARD
```

RFC 6350 所确定的规则定义了 `text/vcard` 文档的语义。你可以按照这些规则对这个文档进行解析进而构造出一个人的大致轮廓——他的名字、出生日期等。

你被束缚在这里了。这些应用语义意义明确，但是却没有任何链接。这个文档对于超媒体来说是个死胡同。

当然，HTTP 的协议语义还是适用的。你发送 GET 请求就会收到这个表述。API 可能允许你修改表述然后 PUT 回去。你可能还可以发送 DELETE 请求来删除所对应的资源。但是你不能从这个表述移动到另外一个相关的资源上去，因为 vCard 不是一个超媒体格式。

由于 vCard 是在电话和软件的通讯录中常用的一种格式，所以将 vCard 表述作为在超媒体迷宫尽头的奖品是有意义的。客户端可以通过超媒体定位一个“person”资源，然后就可以访问一个“export to vCard”链接来获取这个“person”资源的 `text/vcard` 表述。

但是这很可能并不是你想要的。你不想将一个人的基本信息作为“奖品”。它很可能是一个 API 的主要部分。你可能想要抄袭 vCard 的应用语义并将它们应用到一个真正的超媒体文档中。

hCard microformat (<http://microformats.org/wiki/hcard/>) 的设计者正是这么做的。他们并没有重复 vCard 标准的制定者们已经完成的工作，而是让 HTML 这一超媒体文档格式来展示同样的信息成为可能。

我在第 7 章中将对 hCard 进行更多介绍，现在只是一个预览——这是我前面展示过的 vCard 文档的 hCard 版本：

```
<div class="vcard">
  <span class="fn">Jennifer Gallegos</span>
  <span class="bdai">1987-08-25</span>
</div>
```

hCard 微格式使得你可以将一个人的 vCard 样式的表述与超媒体链接以及表单组合起来，从而实现一个完整的 web API。

这也就是在开始设计 API 之前查阅寻找相关领域特定的数据格式非常重要的另一个原因。类似于 vCard 这样的标准都是针对某个问题领域花费了大量的时间和金钱，从而确认了其应用层语义后的成果。你并不需要仅仅因为 vCard 没有超媒体控件就一切都重新来过。

即便你不能直接复用某个领域特定设计，你也可能能够通过将它的应用层语义整合到一个 profile 中来为你节省一些时间。但这将是第 8 章的一个主题。

86 如果找不到相关的领域特定设计，不要自己制造

如果你不能为你的问题领域找到一个合适的领域特定 API 的话，不要慌张。人们并不是经常定义可复用的、领域特定的、支持超媒体的格式。这并不意味着你不得从零开始。你应该能够以某种标准化的格式为基础，对其进行扩展并复用其他人可能已经完成的工作。你需要做的工作并不多，仅仅是将各个部分粘合在一起。

在下面两章中，我将会讨论一些基础内容。需要指出的是，这其中的一些领域特定设计会用来处理一些相当流行和普遍的领域——事物的集合。我也并不是真的认为这是一个领域，它更像是一个设计模式。

API 客户端的种类

除去本章中讲到的 3 个 Maze+XML 客户端，我在本书中也不会再讨论更多关于 API 客户端的内容了。因为现在网上部署的能完全发挥 Fielding 约束的优点的 API 并不多，所以我也无法给予更多的指导。

即便你对于超媒体的知识理解非常深刻，可是在你遇到一个不支持超媒体文档的 API 时，那些知识对于你为这个 API 编写客户端也是毫无帮助的。当你编写客户端时，你是任由服务器设计摆布的，并且实用主义总是打败理想主义。现在，实用主义意味着为每个 API 采用不同的方法。

但是我已经看到过非常多的已经部署的超媒体 API（包括万维网自己），所以我可以讲一讲人们通常编写的客户端的种类。我已经理解：客户端和服务端必须对问题领域有一致的理解，但是它们却并不需要共享同一个目标。老实说，这是我第一次尝试对这些我们所编写的用来完成我们的目标的客户端进行分类。

人类驱动的客户端

人类驱动的客户端可以拥有相对简单的逻辑，这是因为它们并不需要做出任何决定。它们将表述展示给人类，并且将人类的决定回传给服务器。人类驱动的客户端之间的区别取决于它们在多大程度上忠实地将表述展示给它们的人类用户。

标准的 web 浏览器就是一个忠实的渲染程序，几乎网页中的每一个 HTML 标签都会对屏幕上显示的内容产生一些图形影响。HTML 文档中的每个超媒体链接以及表单都会出

现在屏幕上，除非文档中有一些另外的内容声明它们应该被隐藏起来。

现在考虑一下采用文本 - 语音转换技术来向视力受损的用户展示网页的 web 浏览器。这样的浏览器在忠实于原文方面就稍微差了一些。有一些 HTML 标签可以很好地翻译成语音的表现方式（ 就是一个很好的例子），有的就不行（比如 <div>）。但是任何 web 浏览器都必须忠实地渲染超媒体控件。视力正常的用户所能触发的每个链接和表单也应该同样能够被视力受损用户所使用。

游戏客户端就是一个相当忠诚的 Maze+XML 文档渲染器。Maze+XML 文档不像 HTML 文档那样包含布局信息，但是游戏客户端能保证将它所发现的资源状态的所有信息（比如一个单元格的标题）、所有的超链接都展示给它的人类用户。

用只有“左转”、“右转”、“前进”命令的坦克控件来取代方向链接的游戏客户端将是一个不够忠实的渲染器，尽管它向服务器发送的请求和我之前展示给你的游戏客户端发送的请求是一样的。但是一个拒绝将“exit”链接展示给用户并导致用户被永远困在迷宫之中的游戏客户端根本不是一个非常忠实的客户端。

一个不那么忠实的渲染器会在服务器发送的内容和用户体验之间增加一些源于它自身的态度和想法的想象。这听起来很糟糕——没有人会喜欢“不忠实的”事物——但是一件事物忠实与否取决于用户想要做什么。一个商店的网站可能会展示给你很多昂贵的而且你不需要的物品。这时，你可能会更喜欢一个不那么忠实的渲染器：一个根据网店的 API 能自动过滤掉昂贵商品来帮助你找到物美价廉的物品的客户端。

当人类用户选择一个迷宫游玩的时候，这个游戏客户端会获取一个迷宫的表述，但是它并不将这个表述显示出来。它会扫描表述找到 `rel="start"` 的链接，然后自动地访问这个链接。这是“不忠实的”。这个游戏客户端认为在迷宫的表述中没有人类用户感兴趣的内容，并且让人类手动单击“start”链接是浪费时间的行为。这很可能是正确的，但是也正是因此，游戏客户端并不是一个完全忠实的客户端。

客户端为了忠实地渲染收到的表述以及不将自己的判断力牵杂其中而付出的努力越多，它在遇到不期望的表述的时候就越不容易崩溃。

自动化客户端

自动化客户端会收到表述，但是不对它们进行渲染。并没有人类成员来看渲染的结果。客户端通过决定触发哪个超媒体控件来消除它们自己所遇到的语义鸿沟。当然，大部分客户端并不能“决定”任何事情。它们仅仅是贯彻执行简单的预先编程实现的规则集合，这些规则有希望能帮助它们来实现一些预定义的目标。

没有哪个客户端像人一样智能。但是它们却实实在在地能够将我们从重复的、并不需要我们太多智力的任务中解放出来。我见过并且开发过一些不同类型的自动化客户端。

爬虫

爬虫模拟了一个好奇但是并不挑剔的人。给它提供一个 URL 来启动，它会获取到一个表述。然后，它会访问所有它能发现的链接用来获取更多的表述。它会以递归的方式重复这样的工作，直到它再也获取不到更多的表述为止。

本章前面讲到的地图生成器客户端就是某种 Maze+XML 文档的爬虫。搜索引擎所使用的蜘蛛是 HTML 文档的爬虫。

为一个没有采用超媒体的 API 编写爬虫是很困难的。但是你可以为基于超媒体的 API 编写爬虫，而且你在编写爬虫的时候甚至都不需要理解这个 API 的链接关系。

通常来说，爬虫只会触发那些安全的状态转换。否则，没人会告诉你资源状态会发生什么事情。一个向它所遇到的每个资源都发送 DELETE 请求而仅仅是要看发生了什么事情的爬虫是一个非常可怕的客户端。

监视器

监视器客户端是和爬虫完全相反的一种客户端。它模拟的是一个需要观察某个特定网页的人。给它提供一个 URL 用于启动，监视器会获取这个 URL 的表述，并通过某种方式进行处理。但是它不会访问任何链接。相反的是，它会等待一段时间然后再次抓取同一个资源的新的表述。和触发某个超媒体控件来改变资源状态不同，监视器客户端是等待另外一些事物来更改资源状态，然后检查并查看发生了什么事情。

RSS 聚合器就是一种监视器。人类用户将聚合器指向一系列他们感兴趣的 RSS 和 Atom 订阅源。监视器客户端就周期性地获取这些订阅源，并通过某种方式通知用户有新的内容发布。

假设其中有一个 Atom 订阅源连接到一个采用 Atom 发布协议的功能全面的 API。聚合客户端并不会注意到这些。它只是想要查看订阅源。用户对 RSS 聚合器所做的任何操作都不会改变发布这些订阅源的网站的资源状态。

脚本

当今大部分的自动化 API 客户端都是脚本。脚本模拟一个有固定的日常工作并从来不会改变的人。当这个人厌烦了这样的日常工作并想要将这些工作自动化的时候，脚本就出现了。

由人类来选择一套 API，并且指定那些对完成这项日常工作所必须的状态转换（对于支持超媒体文档的 API）或者 API 调用（对于不支持超媒体的 API）是必要的。然后，人类就编写一个算法，这个算法会使触发状态转换或者发起 API 调用的过程自动执行。

本章前面讲到的那个吹牛者客户端就是一个非常简单的脚本。它知道一些信息（迷宫的标题），它还知道哪里可以找到这个迷宫。它会在你每次运行它的时候给你提供不同的结果（因为它每次会选择一个新的迷宫），但是它总是完成同样的任务：假装走出了迷宫。

一个真正进入迷宫并且向东移动三次的客户端会是一个更让人印象深刻的脚本。如果你给它提供一个合适的迷宫，它甚至具有找到出口的能力！但是很明显，这个算法中并没有智能的内容。它只是一个脚本，一个回放一系列预定义的状态转换的脚本。

当脚本背后的假设不再有效的时候，脚本倾向于终止工作。一个只是向东走三步的“迷宫穿越者”的客户端只能穿过极少部分的迷宫。一个从某个网站提取数据的屏幕抓取脚本也会在 HTML 表述被重新设计以后停止工作。

在一些不重要的事情（比如某个资源的 URL 发生变化或者有新的数据要添加到某个表述中）等发生的时候，一个能理解超媒体的脚本不会那么轻易停止工作。这就是说一个超媒体 API 在不阻碍依赖于它的脚本正常运行的情况下有一定的变更空间。但是脚本只是人类思维过程的回放而已。如果它遇到一个人类在起初没有考虑到的情况，脚本将还是不能填补这块空白。

代理机器人

忘记那个吹牛者客户端。忘记向东移动三次然后停止的脚本。想象一个采用像 wall-following 这样的算法能真正依靠自己能力走出迷宫的客户端。这样的客户端将能够走出它以前从来没有见过的迷宫。它会在探路的过程中根据从服务器端收到的表述来改变自己的行为。它会做出决策。^{注 6}

一个软件代理机器人会模拟积极地处理某个问题的人类。它不像人类那样智能，它也没有能力来做出主观的判断，但是它会去做人类在相同情况下会做的事情。它会查看某个表述，分析当前形势，决定激活哪个超媒体控件来帮助自己更加接近自己的终极目标。

监视器客户端不会这样做；它从不激活超媒体控件。爬虫也不会这么做；它会激活每个它所发现的安全的超媒体控件。脚本也不会这么做；它总是激活那个程序中写好的那下一个超媒体控件。人类驱动的客户终端也不会这么做；它们将任务委派给人类。软件代理机器人是唯一能被称为自主决策的客户终端。

注 6 你可以在本页 (<http://amundsen.com/examples/misc/maze-client.html>) 中看到一些类似的 Maze+XML 客户端。

软件代理机器人可以像穿越迷宫机器人那样简单，也可以由复杂的推理引擎通过将许多不同来源的信息进行综合来驱动。现在，软件代理机器人客户端倾向于简单的一侧。但是我们也想象的到一些更加复杂的代理机器人，比如：私人导购、科幻小说的自动化新闻采集器以及现实生活中金融软件所采用的高频率交易算法程序。

软件代理机器人是最能发挥超媒体 API 的灵活性的自动化客户端。但是它基于两个很大的潜在的前提假定：它的目标是合理的；编程实现的推理过程最终会通往这个目标。如果违反了这些假设，代理机器人将会停止工作。如果穿越迷宫的客户端遇到了一个使得它的算法不能工作的迷宫，比如，一个只有单行道的迷宫——客户端就会像那个总是向东走三步的脚本一样停止工作。

软件代理机器人是由计算机程序员编程实现的，我们对计算机将重要的决定留给软件处理的过程了解得一清二楚。在关键时刻，API 客户端可以选择让人类来确认一个不安全的状态转换（“你想要我来给你买下这件衬衫吗？”）或者做出主观的判断（“这些景点中，哪一个更漂亮？”）。在这个时候，代理机器人就变成了一个人类驱动的客户终端，这很可能会减少错误和降低错误发生时所付出的代价。

集合模式 (Collection Pattern)

回顾第2章，我曾向大家展示过一个简单的微博API，它可以提供媒体类型为application/vnd.collection+json的表述。这些表述看上去是这样的：

```
{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/",

    "items" : [
      { "href" :
        "http://www.youtypeitwepostit.com/api/messages/21818525390699506",
        "data" : [
          { "name" : "text", "value" : "Test." },
          { "name" : "date_posted", "value" : "2013-04-22T05:33:58.930Z" }
        ],
        "links" : []
      },

      { "href" :
        "http://www.youtypeitwepostit.com/api/messages/3689331521745771",
        "data" : [
          { "name" : "text", "value" : "Hello." },
          { "name" : "date_posted", "value" : "2013-04-20T12:55:59.685Z" }
        ],
        "links" : []
      },

      { "href" :
        "http://www.youtypeitwepostit.com/api/messages/7534227794967592",
        "data" : [
```

```

        { "name" : "text", "value" : "Pizza?" },
        { "name" : "date_posted", "value" : "2013-04-18T03:22:27.485Z" }
    ],
    "links" : []
}
],
"template" : {
    "data" : [
        { "prompt" : "Text of message", "name" : "text", "value" : "" }
    ]
}
}
}

```

在本章中，我将会讨论更多关于 Collection+JSON^{注1} 的内容，它是用以定义上述文档结构的标准。

Collection+JSON 是众多被设计用于表示非特定问题域（相对地，Maze+XML 是针对特定问题域的）的通用标准中的一员，它符合集合模式，这种模式在各个领域中频繁显现。我们通过该标准构建了一个很好的例子，因为它是那些初出茅庐的 API 设计新手们很容易理解的一个基于 JSON 的 API 的正式版本。Collection+JSON 让你可以遵循内心最自然的设计偏好，而不会与 Fielding 约束发生冲突。

刚才所展示的这个文档代表了一组微博帖子。而购物车中的一组商品或者是来自气象传感器的一组读数看上去是非常相似的，它们都具有非常相近的协议语义。我对 Collection+JSON 标准所添加的唯一元素就是那一点点应用语义。我确定一条微博应该具有一个 `date_posted` 字段和一个 `text` 字段。而购物车中的一件商品和来自气象传感器的一条读数都将会具有不同的字段，这都反映出了它们拥有不同的应用语义。

如果恰巧在你的问题域中还没有领域特定标准（这或许不太可能），你或许可以采用一个基于集合的标准来代替。相比于你从零开始，你可以将你的精力专注于将你的应用语义适应集合模式。这不仅仅节省了你的时间，你同时将会得到一些既有的客户端程序和服务端工具。

虽然本章专注于讨论 Collection+JSON，我同样也会谈及 Atom 发布协议（Atom Publishing Protocol），或者叫作 AtomPub。AtomPub 是基于集合的 API 的原始标准，它的定义见 RFC 5023。它是一种相对比较老的标准，但是除了在谷歌公共 API 中被采用过之外，它并没有流行起来——其中一部分原因是它是基于 XML 格式的，而目前在该领域处于霸主地位的是 JSON 形式的表述。

注1 Collection+JSON 由一份个人标准定义，见 <http://amundsen.com/media-types/collection/>。

在第 10 章中，我将会谈到 OData，它是集合模式中的第三项主要标准。OData 是一项尚在制定中的开放标准，它最初是基于 AtomPub 的。它具备了 JSON 表述的优势，并且由微软提供支持。微软已经在它的 Visual Studio 开发平台中集成了对 OData 的支持。

Hydra 标准（见第 12 章）同样也支持集合模式，虽然这并不是它的主要用途。如果我们在基于集合的 API 方面能拥有一个单一且一致的标准，那将是一件非常不错的事情。但是眼下却有 4 种标准正在互相竞争，即使如此，也总是胜过我们目前所拥有的数千个一次性设计。

什么是集合？

在深入围绕集合模式设计的标准细节之前，让我们先探讨下该模式本身。这其实相当简单，但是我还是想显式地将每件事都罗列出来，所以请不要过于惊讶。

集合是一种特别的资源类型。回顾第 3 章中关于资源的定义，资源是足够重要并提供其自身 URL 的任何事物。一个资源可以是一条数据、一个物理对象或者是一个抽象的概念——甚至是任何东西。所有的重点便是它拥有一个 URL 及其表述——即当客户端向该 URL 发送 GET 请求时所收到的文档。

一个集合资源相比于上述的资源会更加特殊些。它的存在主要是为了将其他资源组合到一起，它的表述主要专注于那些链向其他资源的链接，尽管它也可能包含来自其他资源表述的一些片段（或甚至可能是全部表述！）。



集合是一个以链接方式罗列其他资源的资源。

链向子项的集合

包含在集合中的一个独立的资源通常可以被称为集合的子项 (item)、条目 (entry) 或者是成员 (member)。联想到你朋友手机中的联系人列表，你被展示在这样一个列表里：里面有你的名字和你的手机号码。你是该“联系人列表”集合的一个子项。

但是你不只是集合中的一个子项：你是一个人类。你朋友手机中保存的并不是你，而只是一个指向你的链接（通过你的手机号码）和一些关于你的信息（你的名字）。你拥有一个独立的存在，你朋友手机里的数据只是你的部分表述。

相似的是，一个为集合所描述的资源并不会在突然间就成为了一个被称为“子项”的特

殊事物。该资源仍然拥有自己的 URL 和一个在集合之外的独立存在。当我们在讨论某个“子项”或者“条目”，甚至是“成员”时，实际上是在讨论的是一个恰好被链接到集合表述中的独立资源。

Collection+JSON

现在, 让我们瞧瞧 Collection+JSON 是如何实现集合模式的, 从而获得一些更具体的感受。Collection+JSON 标准定义了一个基于 JSON 的表述格式, 它同样也为 HTTP 资源定义了协议语义, 这些资源会以这样的格式来响应 GET 请求。

下面是一个 Collection+JSON 文档：

```
{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/",

    "items" : [
      { "href" : "/api/messages/21818525390699506",
        "data" : [
          { "name" : "text", "value" : "Test." },
          { "name" : "date_posted", "value" : "2013-04-22T05:33:58.930Z" }
        ],
        "links" : []
      },

      { "href" : "/api/messages/3689331521745771",
        "data" : [
          { "name" : "text", "value" : "Hello." },
          { "name" : "date_posted", "value" : "2013-04-20T12:55:59.685Z" }
        ],
        "links" : []
      }
    ],
    "links" : [
      { "href" : "/logo.png", "rel" : "icon", "render" : "image" }
    ],

    "queries" : [
      { "href" : "/api/search",
        "rel" : "search",
        "prompt" : "Search the microblog archives",
        "data" : [ { "name" : "query", "value" : "" } ]
      }
    ]
  },
}
```

```

    "template" : {
      "data" : [
        { "prompt" : "Text of message", "name" : "text", "value" : "" }
      ]
    }
  }
}

```

一个对象基本上都具有 5 个特定属性，这是为应用特定的数据所预定义的数据槽 (predefined slots)：

95

href

一个指向集合本身的永久链接。

items

包含指向集合成员的链接，以及它们的部分表述。

links

指向其他与集合相关资源的链接。

queries

用于搜索集合的超媒体控件。

template

用于向集合添加子项的超媒体控件。其中还有一个可选的用于错误消息的错误区域，但是我在此处的讨论中将不会涉及到。

子项的表示

让我们聚焦子项，这是 Collection+JSON 表述中最重要的一個字段：

```

  "items" : [
    { "href" : "/api/messages/21818525390699506",
      "data" : [
        { "name" : "text", "value" : "Test." },
        { "name" : "date_posted", "value" : "2013-04-22T05:33:58.930Z" }
      ],
      "links" : []
    },
    { "href" : "/api/messages/3689331521745771",
      "data" : [

```

```

    { "name" : "text", "value" : "Hello." },
    { "name" : "date_posted", "value" : "2013-04-20T12:55:59.685Z" }
  ],
  "links" : []
}
]

```

我之所以说它是最重要的字段，是因为它让我们清晰地了解到集合中有哪些子项。在 Collection+JSON 中，每个成员都被表示成一个 JSON 对象。像集合本身一样，每个成员都具有一定数量的预定义数据槽，它们可以被应用特定的数据填充：

href属性

一个指向子项的永久链接，这个子项被作为独立的资源看待。

96

links

指向子项相关的其他资源的超媒体链接。

data

任何其他信息，这是子项表述的一个重要部分。

子项的永久链接

成员的 href 属性是一个指向它所在集合环境之外的资源的链接。如果你向 href 属性中的 URL 发起 GET 请求，服务器将会向你发送该单个子项的 Collection+JSON 表述。它看上去将会像下面这样：

```

{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/",

    "items" : [
      { "href" : "/api/messages/21818525390699506",
        "data": [
          { "name" : "text", "value" : "Test." },
          { "name" : "date_posted", "value" : "2013-04-22T05:33:58.930Z" }
        ],
        "links" : []
      }
    ]
  }
}

```

你或许可以通过向其永久链接发送 HTTP 的 PUT 请求来修改这个子项，或者是通过

HTTP 的 DELETE 请求来删除它，这些都是子项的协议语义。它们作为 Collection+JSON 中关于“item”定义的一部分，讲得非常清楚。

子项的数据

任何 Collection+JSON 应用的核心都是你尝试传达的应用级语义：即那些与每个独立子项关联的数据。大部分的数据都进入了子项的 `data` 数据槽，该数据槽需要包含一个 JSON 对象的列表，每个对象都具有 `name` 和 `value` 属性，并都被描述为单个键值对。下面是一个来自我们的微博 API 的例子：

```
"data" : [
  {
    "name" : "text",
    "value" : "Test.",
    "prompt" : "The text of the microblog post."
  },
  {
    "name" : "date_posted",
    "value" : "2013-04-22T05:33:58.930Z",
    "prompt" : "The date the microblog post was added."
  }
]
```

97

其中的 `name` 属性是该键值对的键，而 `value` 当然就是值了，还有一个（可选的）`prompt` 值是一段人类可读的描述信息。Collection+JSON 标准并没有说明你应该用什么样的键、值以及提示，这取决于你的需求以及你为 API 定义的应用级语义。

子项的链接

Collection+JSON 中最简单的超媒体控件便是 `href` 属性。我在上文中曾有所提及，它是一个特定的链接，客户端可以在想要引用一个特定的子项的时候随时使用链接所提供的 URL。

```
"href" : "/api/messages/21818525390699506"
```

子项的表述可以同样包含一个称为 `links` 的列表。它包含了任意数量的指向相关资源的其他的超媒体链接。下面是一个你可能会在一个“图书”资源的表述中看到的链接：

```
{
  "name" : "author",
  "rel" : "author",
  "prompt" : "Author of this book",
  "href" : "/authors/441",
  "render" : "link"
}
```


它近似等价于下面的 HTML 片段：

```
<a href="/authors/441" id="author" rel="author">Author of this book</a>
```

rel 属性是一个为链接关系准备的数据槽，就好比 Maze+XML 中的 rel 属性。它是一个可以让你放置某些应用语义的地方。而 prompt 属性是一个用于放置人类可读的描述信息的地方，这就好比是 HTML 的 <a> 标签中的链接文本。

下面是另一个你可能在一段图书表述中看到的链接：

```
{
  "name" : "cover",
  "rel" : "icon",
  "prompt" : "Book cover",
  "href" : "/covers/1093149.jpg",
  "render" : "image"
}
```

它近似等价于下面的 HTML 片段：

```

```

author 链接和 icon 链接之间的区别在于 render 属性。如果将 render 属性设置为 link，则等于告诉 Collection+JSON 客户端应该将该链接呈现为一个转出的链接（见第 4 章），就好比 HTML 的 <a> 标签。用户可以单击该链接，从而将客户端的视图转向另一个表述。当 render 属性设置为“image”时，等于告诉客户端应该将该链接呈现为一张内嵌的图片，就好比 HTML 的 标签。该链接的表述会被自动抓取，并且该表述将会被直接整合到当前表述的视图中。

写入模板（Write Template）

假设你想要向集合添加一个子项，那你应该发起什么 HTTP 请求呢？为了回答这个问题，你需要看看集合的写入模板。

下面是为我们的微博 API 准备的写入模板：

```
"template": {
  "data": [
    {"prompt" : "Text of message", "name" : "text", "value" : ""}
  ]
}
```

根据 Collection+JSON 标准对模板进行解释，解释完没有问题之后你便可以填写该模板的空白处，然后提交一个如下的文档：

```
{ "template" :
{
  "data" : [
    { "prompt" : "Text of the message", "name" : "text", "value" : "Squid!" }
  ]
}
}
```

该请求会去向何方？Collection+JSON 标准中提到你需要通过向某个集合（也就是向它的 href 属性中的 URL）发送一个 POST 请求才能向该集合添加一个子项：

```
"href" : "http://www.youtypeitwepostit.com/api/",
```

因此，该 POST 请求看上去将会是如下的样子：

```
POST /api/ HTTP/1.1
Host: www.youtypeitwepostit.com
Content-Type: application/vnd.collection+json

{ "template" :
{
  "data" : [
    { "prompt" : "Text of the message", "name" : "text", "value" : "Squid!" }
  ]
}
}
```

这意味着写入模板概念上等同于如下的 HTML 表单：

```
<form action="http://www.youtypeitwepostit.com/api/" method="post">
  <label for="text">Text of the message</label>
  <input id="text"/>
  <input type="submit"/>
</form>
```

99

它们并不完全相同，因为填写完的 HTML 表单发送的是一个 application/x-www-form-urlencoded 的表述，而填写完的写入模板发送的是一个 application/vnd.collection+json 表述。但是从概念上来说，这两种超媒体控件是非常相似的。

搜索模板

如果一个集合拥有数百万的子项，那么服务器为每个客户端的 GET 请求都发送所有子项的表述将是一种非常愚蠢的行为。服务器可以通过提供搜索模板来避免这种情况——客户端可以通过填写超媒体表单来对一个 Collection+JSON 集合进行过滤。

一个集合的搜索模板被保存在 queries 数据槽中，下面的 queries 数据槽包含了一个简单的搜索模板：

```
{
  "queries" :
  [
    {
      "href" : "http://example.org/search",
      "rel" : "search",
      "prompt" : "Search a date range",
      "data" :
      [
        { "name" : "start_date", "prompt": "Start date", "value" : "" },
        { "name" : "end_date", "prompt": "End date", "value" : "" }
      ]
    }
  ]
}
```

Collection+JSON 搜索模板等价于下面的 HTML 表单：

```
<form action="http://example.org/search" method="get">
  <p>Search a date range</p>
  <label for="start_date">Start date</label>
  <input label="Start date" id="start_date" name="end_date" value=""/>

  <label for="end_date">End date</label>
  <input label="End date" id="end_date" name="end_date" value=""/>
</form>
```

同样也等价于下面的 URI 模板：

```
http://example.org/search{?start_date,end_date}
```

我之所以说它们都是等价的，是因为这 3 种方式根据相同的输入都会产生相同的 HTTP GET 请求。该请求看上去将会是像下面这样的：

100

```
GET /search?start_date=2010-01-01&end_date=2010-12-31 HTTP/1.1
Host: example.org
```

一个（通用的）集合是如何工作的

关于 Collection+JSON 的内容我已经展示得够多了。它的设计中并没有掺杂任何实际的应用语义，所以它可以在很多不同的应用中被使用。由于它如此通用，从而很好地勾画出了集合模式的公共特性。

在我们开始转向 AtomPub 的讨论之前，我想先对 HTTP 下的一个普通的“集合”资源的行为进行描述，进而上升一个层次来对我所认知的模式展开介绍。Collection+JSON、AtomPub、OData 以及 Hydra 对集合都采用了不同的方式，但是它们都或多或少具有相似的协议语义。

GET

和大多数资源一样，集合在响应 GET 请求时会提供一个表述。虽然这 3 种主要的集合标准都没有详细指明集合中的子项应该是什么样子的，但是它们都非常具体地说明了集合的表述应该是怎样的。

表述的媒体类型告诉了你可以对资源作何处理。如果你得到的是一个 `application/vnd.collection+json` 表述，你将可以应用 Collection+JSON 标准的规则。而如果表述是 `application/atom+xml` 格式的，你就会知道应该应用 AtomPub 的规则。

如果表述是 `application/json` 格式的，你就没那么幸运了，因为 JSON 标准并没有说明任何关于集合资源的内容。你正在使用一个自生自灭的独自维持的 API、一个 fiat 标准的 API。你将需要对你所使用的这个 API 进行更加仔细的审视。

POST-to-Append

集合的典型特征就是它在 HTTP POST 请求下的行为。除非集合是只读的（比如搜索结果的集合），否则客户端就可以通过 POST 请求向该集合添加子项。

当你向集合 POST 一个表述时，服务器便会基于你的表述创建一个新的资源，该资源将会成为这个集合最新的成员。回顾第 2 章，当向微博 API 发送 POST 时，一条新的帖子将会“插入”到该微博中。

PUT和PATCH

主要的集合标准都没有对集合在应答 PUT 和 PATCH 请求时的响应进行定义。一些应用通过实现这些方法来一次性修改集合中的多个元素，或是从集合中删除个别元素。

Collection+JSON、AtomPub 和 OData 都为子项定义了应答 PUT 方法的响应：它们认为 PUT 方法是客户端用以改变子项状态的方式。但是这些标准只是重复了 HTTP 标准中规定的内容，它们并没有为子项资源添加新的约束。PUT 是客户端用以改变任意 HTTP 资源状态的方式。

DELETE

3 个主要标准都没有定义集合该如何响应 DELETE 请求。一些应用通过删除集合来实现 DELETE 方法，而另一些则会删除掉集合以及在集合中罗列的所有子项所对应的资源。

几个主要的集合标准都对子项如何响应 DELETE 进行了定义，但是，它们也只是重述了 HTTP 标准中所陈述的内容。DELETE 方法用于删除事物。

分页

一个集合有可能包含数百万的子项，但是服务器并没有义务在单个文档中提供数百万的链接。而最普遍的替代方案便是分页。服务器可以选择提供集合中的前 10 个子项，并且给客户端提供一个指向剩余项的链接：

```
<link rel="next" href="/collection/4iz6"/>
```

“next” 的链接关系已经在 IANA 进行了注册，它的意义是“该连续内容的下一部分”，通过访问该链接你将可以获取到集合的第 2 页内容。你将可能无限次地访问 `rel="next"` 链接，直到你达到该集合的末尾。

目前有很多用于在分页列表中导航的通用链接关系，它们包括“next”、“previous”、“first”、“last”以及“prev”（“previous”的同义词）。这些链接关系原本是为 HTML 定义的，但是现在它们都在 IANA 上注册过，所以你可以在任何媒体类型中使用它们。

某些基于集合的标准明确地定义了分页技术，而其他的则简单地假定你知道“next”和“previous”。Collection+JSON 便属于后者。它没有明确提供对分页的支持，但是你可以通过将它的通用媒体链接与 IANA 的通用链接关系结合来得到这项功能：

```
"links" : [ {
  "name" : "next_page",
  "prompt" : "Next",
  "rel" : "next",
  "href" : "/collection/page/3",
  "render" : "link"
},
{
  "name" : "previous_page",
  "prompt" : "Back",
  "rel" : "previous",
  "href" : "/collection/page/1",
  "render" : "link"
}
]
```

搜索表单

集合模式最后一项共有的特性是超媒体搜索表单。这同样也对那些非常大的集合带来了很大的帮助。搜索表单可以让客户端在无须下载集合全部内容的情况下找到自己感兴趣的部分。

Collection+JSON 和 OData 明确定义了它们的超媒体搜索表单格式。我曾在本章前面的内容中向你展示过 Collection+JSON 的搜索模板，而 AtomPub 并没有提供对搜索的原生支持，它假定你在需要这项特性时，会自行附加另一个标准，例如 OpenSearch。

Atom发布协议（AtomPub）

当时之所以开发 Atom 文件格式是为了将它作为 RSS 的一种可选方案，用于聚合新闻文章和博客。该文件格式由 RFC 4287 定义，并在 2005 年定稿。Atom 发布协议是一个用于编辑和发布新闻文章的标准化工作流，它使用 Atom 文件格式来作为表述格式。该协议由 RFC 5023 定义，于 2007 年定稿。在 REST API 的世界里，这些都属于非常早期的时间点。事实上，AtomPub 是第一个用于描述集合模式的标准。

下面是一个微博的 Atom 表述，此前我曾使用 Collection +JSON 格式展示过同一条微博。AtomPub 具有和 Collection+JSON 一样的概念，但是使用了不同的术语。相对于包含了“item”的“collection”，下面所采用的是包含了“entry”的“feed”。

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>You Type It, We Post It</title>
  <link href="http://www.youtypeitwepostit.com/api" rel="self" />
  <id>http://www.youtypeitwepostit.com/api</id>
  <updated>2013-04-22T05:33:58.930Z</updated>

  <entry>
    <title>Test.</title>
    <link
      href="http://www.youtypeitwepostit.com/api/messages/21818525390699506" />
    <link rel="edit"
      href="http://www.youtypeitwepostit.com/api/messages/21818525390699506" />
    <id>http://www.youtypeitwepostit.com/api/messages/21818525390699506</id>
    <updated>2013-04-22T05:33:58.930Z</updated>
    <author><name/></author>
  </entry>

  <entry>
    <title>Hello.</title>
    <link
      href="http://www.youtypeitwepostit.com/api/messages/3689331521745771" />
    <link rel="edit"
```

103

```

    href="http://www.youtypeitwepostit.com/api/messages/3689331521745771" />
    <id>http://www.youtypeitwepostit.com/api/messages/3689331521745771</id>
    <updated>2013-04-20T12:55:59.685Z</updated>
    <author><name/></author>
  </entry>

  <entry>
    <title>Pizza?</title>
    <link
      href="http://www.youtypeitwepostit.com/api/messages/7534227794967592" />
    <link rel="edit"
      href="http://www.youtypeitwepostit.com/api/messages/7534227794967592" />
    <id>http://www.youtypeitwepostit.com/api/messages/7534227794967592</id>
    <updated>2013-04-18T03:22:27.485Z</updated>
    <author><name/></author>
  </entry>

</feed>

```

该文档以媒体类型 `application/atom+xml` 的格式提供，并容许 AtomPub 客户端对该文档做出某些特定的假设。你知道你可以向集合的 `href` 发起 POST 请求从而向集合添加一个新的 Atom 条目。如果你想编辑该条目，你可以向条目的 `rel="edit"` 链接指定的 URL 发送 PUT 请求。又或者你想删除该条目则可以向该 URL 发送一个 DELETE 请求。

这些都没有什么让人感觉新奇的，这与 Collection+JSON 的方式非常相似，而且大部分都是重述了 HTTP 标准中的一些思想。

Collection+JSON 和 AtomPub 在概念上有个很大的差别。在 Collection+JSON 中并没有给“item”定义特别的应用语义，一个“item”看上去可以是任何事物。但是因为 Atom 是被设计用来聚合新的文章的，所以每个 AtomPub 条目看上去就像一篇新的文章。AtomPub feed 中的每个条目都必须拥有一个唯一的 ID（我使用了该微博的 URL 作为 ID）、一个标题（我使用了该微博的内容文本），以及它发布或最近一次更新的日期和时间。Atom 文件格式为新闻报道定义了一些应用语义：比如“副标题（subtitle）”和“作者（author）”这样的字段。Collection+JSON 则没有定义任何这些内容，它甚至没有要求集合的每个成员都必须具备一个永久链接（虽然你真的应该让它具备一个）。

104

尽管专注于新闻和博客文章，AtomPub 仍然是一个集合模式的完整的通用实现。谷歌是 AtomPub 最大的企业采用者，它使用 Atom 文档来展示视频、日历事件、电子表格的单元格以及地图上的地点等。

AtomPub 之所以被谷歌选择的秘密在于其可扩展性。它允许你定义任何你所关心的应用语义来扩展 Atom 的词汇表。谷歌为它所有基于 Atom 的 API 定义了一个公共的 Atom 扩展，称为 GData，并且又为视频、日历、电子表格等定义了额外的扩展。

一些关于 AtomPub 在遵守集合模式方面的有趣事实：

- 因为新闻文章通常被归类到一或多个分类下，所以 Atom 文件格式定义了一个简单的分类系统，并且 AtomPub 为分类列表单独定义了一种媒体类型（application/atomcat+xml）。
- AtomPub 同样为服务文档（Service Document^{注2}，即集合的集合）定义了一种媒体类型。
- Atom 是一种严格的基于 XML 的文件格式。AtomPub 这套技术并不能提供 JSON 表述，这让 Ajax 客户端很难使用 AtomPub API。谷歌认识到了这个问题，并在提供 AtomPub 表述的同时，为文档添加了 JSON 表述。但是谷歌只是将它作为一种 fiat 标准，而不是一项鼓励大家复用的标准。
- 尽管 Atom 是一种 XML 的文件格式，客户端也可以向一个 AtomPub 的 API POST 二进制文件。一个在服务器端的上传文件将被表示为两个不同的资源：一个是媒体资源，其表述是一组二进制数据；另一个是 Entry 资源，它的表述是 Atom 格式中的元数据。该特性允许你在使用 AtomPub 保存包含了描述信息和相关链接的 Atom 文档的同时，也将你的一组照片或音频文件存储起来。

AtomPub 插件标准

因为它们非常容易扩展，所以 Atom 和 AtomPub 经常被用来作为很多小型插件标准的基础，从而增强集合模式：

- Atom 线索扩展（Atom Threading Extensions，由 RFC 4685 定义）可以更加容易地描述常见于邮件线索和留言板的会话结构。这个扩展并不大——仅仅包含一些额外的标签和一个称为“replies”的新的链接关系。
- Atom deleted-entry 元素（由 RFC 6721 定义）让服务器为那些从集合中被删除的子项建立一个“墓碑”，而不是简单地移除它们。这等于告诉客户端需要去清除被删除的条目，而不是一直被缓存着。
- RFC 5005（“Feed 分页和归档（Feed Paging and Archiving）”）定义了“归档 feed”的概念，这是一种对跨多个资源的大型 feed 更加有效的分页方式。它还定义了“next-archive”、“prev-archive”和“current”3 个链接关系来替代“next”、“prev”和“first”。
- OpenSearch（<http://www.opensearch.org/Specification/OpenSearch/1.1>）是为基于 XML 的搜索协议而制定的联合标准。一个 OpenSearch 文档等价于一个 HTML 表单，也等价于 Collection+JSON 文档中的“queries”片段。客户端填写完表单可以执行一次搜索（通过 HTTP 的 GET 请求）并获取一份搜索结果的 Atom feed。

105

注2 关于服务文档的内容见 <http://bitworking.org/projects/atom/rfc5023.html#appdocs>。——译者注

OpenSearch 定义了一个新的链接关系“search”，通过该链接关系可以将 Atom feed 链接到一个 OpenSearch 文档。OpenSearch 并不是特定于 Atom 的，你的 web 浏览器的搜索框也可以使用 OpenSearch。OpenSearch 让你可以搜索不同的网站，无须真正去访问那些网站和使用它们的 HTML 搜索引擎。我之所以将 OpenSearch 纳入本章，是因为 AtomPub 并没有定义搜索协议，但这又是你应该会使用到的。在第 10 章中，我将会对 OpenSearch 有更详细的讨论。

- PubSubHubbub (<http://code.google.com/p/pubsubhubbub/>) 是一个企业标准，它描述了一个发布订阅的协议，该协议可以让客户端进行注册，并在 Atom feed 更新时接收到相应的通知。它定义了一个新的链接关系：“hub”。

所有由这些插件标准定义的链接关系都在 IANA 进行了注册。这意味着“replies”、“next-archive”、“prev-archive”、“current”、“search”和“hub”都是通用的关系，可以在任何地方使用而无须特别说明。其中“search”链接关系是由 OpenSearch 定义的，但是 `rel="search"` 并不是说“这是一个指向 OpenSearch 文档的链接”，它是指“这是一个指向某个类型的搜索表单的链接”。

即使你现在没有使用 AtomPub，你仍可以从那些多年致力于 Atom 扩展的人的工作成果中受益。他们为很多常用操作创建了标准的词汇表，你只需要选择是否要复用它。

为什么不是每个人都选择使用 AtomPub?

在 RFC 定稿到现在已经六年有余，尽管还有各种各样的插件标准，但是肯定地讲，AtomPub 并没有流行起来。除了谷歌之外，该标准没有得到任何来自其他公司的推动，甚至连谷歌看起来也正在逐步淘汰该标准。AtomPub 到底哪里出了问题？

这个问题要追溯到 2003 年时的一个技术决策：AtomPub 的表述被设计为 XML 文档。这在 2003 年看来是个非常明智的决定，但是在接下去的 10 年里，浏览器内的 API 客户端变得越来越流行，而 JSON 这种表述格式获得了势不可挡的人气。对于浏览器中的 JavaScript 代码来说，处理 JSON 比处理 XML 容易太多了。时至今日，绝大多数的 API 都只提供 JSON 格式的表述，或者是在 XML 和 JSON 表述两者间提供选择。AtomPub 几乎已经销声匿迹了^{注 3}。

那么为什么我们还要在本书中给 AtomPub 专门开辟了这么一大段的章节呢？部分原因是该标准本身并没有什么问题，就它的原理来说它工作得很不错。它的历史意义在于它是“集合”API 模式的首个通用实现。它的插件标准所定义的 IANA 注册的连接关系可以在其他的表述格式中被干净地复用。

注 3 Joe Gregorio 是 Atom 和 AtomPub 的主要贡献者，他在博文中 (<http://bitworking.org/news/425/atompub-is-a-failure>) 也提到了相同的情况。

但是 AtomPub 的故事同样也说明了一个道理：“标准没有什么问题”还不够好。人们都很怕麻烦，他们不会刻意去学习一个标准，除非它契合了他们的需求。采用一个基于 JSON 的 fiat 标准来重新实现“集合”模式会更加容易些，这也是数以千计的开发者曾经做过并还在继续这样做着事情。

我写本书的主要目标就是为了终结这种重复的劳动。我不知道答案会不会是 Collection+JSON，又或者是我在后续章节中将会提到的任何其他的超媒体格式。很可能答案并不唯一。

我知道“集合”模式已经证明了它的优势。问题在于我们是否还会允许我们自己一遍遍地重复发明轮子。

语义挑战：我们应该怎么做？

请牢记，语义的挑战是：“我们该如何编写程序让计算机来决定单击哪个链接呢？”为了回答这个问题，我们必须在 HTTP 的协议语义（由 URL 来标识并响应 GET 和 PUT 等方法的一般性“资源”）和你特定、独特的 web API 的应用语义（微博服务、支付处理或任何你正在做的事情）之间建立起桥梁。

像 Maze+XML 这样的领域特定设计通过专门设计的超媒体类型，以及为你的问题空间特别设计的链接关系来为语义鸿沟架起桥梁的。但是这需要做很多的工作，而且几乎没有人能在这项工作上走得很远。

集合模式识别出了两种不同类型的资源：子项类型的资源（通常会响应 GET、PUT 和 DELETE 请求）以及集合类型的资源（通常会响应 GET 和 POST-to-append 请求）。一个集合类型的资源包含了一定数量的子项类型资源，它的表述链接到这些子项，并包含了它们的部分表述。

集合和子项之间的差异形成了 HTTP 协议语义之上的应用语义的边界层。Collection+JSON、AtomPub 和 OData 在集合和子项之间都定义了相同的差异点。因为这些区别的存在，IANA 的很多链接关系顿时便有了意义：用于对集合进行导航的关系，比如“first”、“next”以及“next_archive”；“search”关系用于对集合进行搜索；“item”关系用于指向集合中的一个子项；“edit”关系用于编辑一个子项；而“collection”关系将一个子项和包含它的集合连接起来。

但是就一个“item”而言并没有什么特殊的，它几乎就是一个像“资源”一样笼统和不明确的术语。在一个微博 API 中，一个“item”只有少量的文本和一个时间戳。在一个支付处理器中，一个“item”将包含贷方、借方、支付的方式以及金额的总数。在集合

◀ 107

模式的应用语义和你自有的 API 的应用语义之间还是存在着巨大的鸿沟的。

再看一眼下面微博的 Collection+JSON 表述：

```
{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/",

    "items" : [
      {
        "href" :
          "http://www.youtypeitwepostit.com/api/messages/21818525390699506",
        "data" : [
          {
            "name" : "text",
            "value": "Test.",
            "prompt" : "The text of the microblog post."
          },
          {
            "name" : "date_posted",
            "value": "2013-04-22T05:33:58.930Z",
            "prompt" : "The date the microblog post was added."
          }
        ]
      }
    ]
  }
}
```

HTTP 告诉我们如何来编辑该 item：以某种方式改变表述并将它 PUT 回去。Collection+JSON 则告诉我们表述应该是什么样子的。它应该看上去像一个被填写完整的 Collection+JSON “模板”：

```
PUT /api/messages/21818525390699506 HTTP/1.1
Host: www.youtypeitwepostit.com
Content-Type: application/vnd.collection+json
"template" : {
  "data" : [
    { "prompt" : "Text of message", "name" : "text", "value" : "The new value" }
  ]
}
```

但是 Collection+JSON 并没有说明“text”和“date_posted”的意义。为了解理解这些事物，人们必须阅读在“prompt”元素中那些人类可读的说明。这便是 Collection+JSON 用以为语义鸿沟架起桥梁的方式。Maze+XML 处理语义鸿沟的方式是提前在媒体类型的规范中对应用语义进行定义。而 Collection+JSON 将应用语义写入散落于表述各处的

“prompt”元素。

如果每个人都在它们的 API 中使用 Collection+JSON,我们将可以共享“集合”的共同定义。但是将可能会出现 57^{注4} 种关于“item”的不同定义,57 套不同“prompt”值的数据元素。一些 API 会将文本字段称为“text”,而其他的则可能将它称为“content”或“post”或是“blogPost”,但其实它们都在使用不同的单词描述相同的事物。我们仍将拥有 57 种不同的微博 API。

所以还没有达到我们的目的,我们仍然需要更多别的东西。

注4 此处的 57 源自前言中提到的 ProgrammableWeb (<http://www.programmableweb.com/>),它收录了 57 种微博 API。——译者注

纯-超媒体设计

集合模式的功能很强大，但是它不具有普适性。从技术上说，第 5 章中的迷宫游戏客户端可以采用 Collection+JSON 格式的表述来实现，但是这看起来很糟糕。游戏客户端的意义在于客户端每次只能看到一个单元格。在这其中并不存在需要“集合”到某个集合中的东西。迷宫游戏的应用语义和集合模式所提供的语义并不一致。

没有人要求你必须使用集合模式，但是对于 API 而言，它是最流行的设计模式。如果你想要实现一些其他的模式，或者如果你的 API 设计方案并不符合任意一种特别的模式，你可以使用纯超媒体（pure hypermedia）来描述 API 的语义。你并不需要创造一种类似于 Maze+XML 的拥有自己的媒体类型的崭新标准。你可以用一种通用的超媒体语言（generic hypermedia language）来表示你的资源的状态。

在本章中，我将对一些采用通用的超媒体语言作为它们的表述格式的 API 展开讨论。我会讲到很多新型的表述格式，但是我讲解的重点将是你已经很熟悉的很老的格式：HTML。

为什么是HTML？

我们要以万维网（由无数的供人类阅读的文档组成的网络）为背景来认识 HTML。由于 HTML 的广为流行，只要 API 中有任何一部分是提供供人类消费的文档的，HTML 都将是显而易见的选择。即便你的 API 的其他部分提供的都是基于 XML 或者 JSON 的表述，对于需要呈现给用户的那部分，你还是可以使用 HTML 作为表述的。HTML 如此流行以至于每个现代操作系统都会在发布时捆绑一款用于调试基于 HTML 的 web API 的工具：web 浏览器。

即便是对于那些定位于仅供机器调用的 API 而言，HTML 也具有显著的优势。和 XML 以及 JSON 相比，HTML 对文档施加了更多的结构信息，但是它又不像 Maze+XML 那样结构信息太多以至于只能用于解决某个特定问题。HTML 处于中间层，类似于 Collection+JSON。

不同于一无所有的 XML 和 JSON，HTML 打包了一整套标准化的超媒体控件。但是 HTML 的控件非常通用，并且没有和某个特定的问题空间绑定。Collection+JSON 为搜索查询定义了一个特殊的超媒体控件；而 HTML 则定义了一个可以用于任何目的的超媒体控件（<form> 标签）。

最终，出现了一种流行的说法。那就是，HTML 是迄今为止世界上最流行的超媒体格式。如今已经出现了许许多多的 HTML 解析和生成工具，并且大部分的开发人员都知道如何读取一个 HTML 文档。HTML 如此流行，使得它成为了人们为了消除语义鸿沟而正在做出的两项巨大努力的基础标准，这两项工作分别是：微格式（microformats）和微数据（microdata），我将会在本章后面部分进行介绍。

HTML 的能力

110 HTML 被设计用来展示文本文档的嵌套结构。任何一个 HTML 标签都可以包含由文本内容和其他标签组成的组合体：

```
<p>
  This 'p' tag contains text
  <a href="http://www.example.com/"> and a link </a>.
</p>
```

这个文档并不符合任何数据结构（英文句子也几乎不符合任何的数据结构）。但是，如同你在 JSON 文档中发现的数据结构那样，HTML 文档是可以包含相同的数据结构的。排序的列表可以使用 标签，键值对的集合可以用 <dl> 标签（被称为“dl”是由于 HTML 将这种数据结构称为“definition list”）。

HTML 同样支持未排序的列表（ 标签）、二维数组（<table> 标签）以及不考虑标准的数据结构而将标签进行随意分组的方式（使用 <div> 和 标签）。

超媒体控件

更重要的是，HTML 拥有内建的超媒体控件。我在第 4 章中曾提到过这些控件，下面再次对一些最重要的超媒体控件做简要的讲解：

- <link> 标签和 <a> 标签是简单的转出链接，它们就像 Maze+XML 中的 <link>

标签一样。它们会命令客户端来向某个指定的 URL 发起一次 GET 请求用以获取一个表述。这个表述之后就会变为当前视图。

- `` 标签和 `<script>` 标签是内嵌链接。它们命令客户端自动地向另一个资源发起 GET 请求，并将该资源的表述内嵌到当前视图中。`` 标签表示将收到的表述作为一幅图片嵌入；`<script>` 标签表示将收到的表述作为代码执行。HTML 同时还定义了一些其他类型的内嵌链接，但是上面这两个是最主要的。
- 当 `<form>` 标签将字符串“GET”作为它的 `method` 属性（也就是 `<form method="GET">`）的时候，它充当了一个模板化的转出链接。它会像 URI 模板以及 `Collection+JSON` 中的查询数据槽那样工作。服务器为客户端提供一个基础 URL 以及若干用于输入的字段（HTML `<input>` 标签）。客户端会为这些字段插入值，并将这些值与基地址合并来组成一个独一无二的 URL，然后向这个 URL 发起一次 GET 请求。
- 当 `<form>` 标签使用“POST”作为它的 `method` 属性时，它所描述的就是一个能够做任何事情的 HTTP POST 请求。那些 `<input>` 标签还是存在的，但是它们不再用于生成请求 URL，而是用于创建一个媒体类型为 `application/x-www-form-urlencoded` 的实体消息体。请求 URL 是在 `<form>` 标签的 `action` 属性中写死的。

应用语义插件

HTML 为一个非常通用的应用（人类可读的文档）定义了应用语义。HTML 标准为相片、标题、章节、列表以及其他在新闻和书籍中常见的结构化元素都定义了标签。

HTML 没有为迷宫或者迷宫中的单元格定义标签。那不是它的应用范围。但是 HTML 不同于 `Maze+XML` 和 `Collection+JSON` 的地方就在于我们可以很容易地在 HTML 的应用范围之外的地方使用它。HTML4 定义了三个通用的属性，我们可以使用它们来添加一些 HTML 标准没有定义的应用语义（HTML5 定义的更多，我将在后面章节中介绍）。

rel 属性

HTML 的 `<a>` 和 `<link>` 标签都有一个称为 `rel` 的属性，这个属性用来定义所链接的资源与当前资源的关系。我们之前已经见到过 `rel`：

```
<link rel="stylesheet" type="text/css" href="/my_stylesheet.css"/>
```

这段 HTML 代码是说 `/my_stylesheet.css` 资源被获取后应该自动地应用到当前页面上以修改其页面样式。在这里，HTML 的 `<link>` 标签是作为一个内嵌的链接而提供服务的。当 `<link>` 标签的 `rel` 使用不同的值（比如，`rel="self"`）时，它就是作为一个转出

链接^{注1}而提供服务了。

112 尽管有一些关于链接关系的标准列表（就像我在第 5 章中提到的 IANA 注册表），但是这些字符串“stylesheet”和“self”也并没有什么特殊之处。某些人为了增强 HTML 而制定了这些字符串。如果你想发布一个 HTML 格式的迷宫 API，你可以继续在 HTML 文档中采用 Maze+XML 中定义的链接关系（“north”、“south”等）。这会为 HTML 格式提供一些它原本并不拥有的应用层语义：迷宫以及迷宫中的单元格的语义。你也可以创建扩展链接关系（看起来像 URL 的链接关系）来描述你的应用中不同资源之间的特定关系。

制定你自己的链接关系的一个缺点就是：你的用户将不清楚那些链接关系的含义，你将在某个 profile 中将这些应用语义记录下来（见第 8 章）。

id 属性

几乎任何 HTML 标签^{注2}都可以为 id 属性赋值。这个属性唯一地标识了文档中的一个元素：

```
<div id="content">
```

如果你碰巧正在找一个 id="content" 的标签，很好，上面这个就是了。一个 HTML 文档中不能包含两个拥有相同 ID 的元素。

我不建议使用 id 属性来和你的应用层语义产生关联。在一个文档中 IDs 必须唯一的要求太苛刻了。这会使得在两个 HTML 文档定义了相同的 id 的情况下，我们不能够将这两个文档合并成一个更大的文档。

class 属性

几乎每一个 HTML 标签^{注3}都可以为 class 属性赋值。这是 HTML 中最为灵活的一个语义属性。在万维网中，class 通常被用于应用 CSS 格式化，但是它同样可以用于传递某些标签的应用语义；按照字面意思，它属于什么“class”。

下面是一个简单的例子，它是一个 <div> 标签，但是它包含了两个 标签：

注 1 HTML4 同样允许链接拥有 rev 属性，它是 rel 的反义词。rev 的值所代表的是这个资源与所链接的资源的关系。在指向下一页的链接中，rel 应该是 next，而 rev 应该是 previous。这也就证明了 rev 属性不是必要的。而且它已经从 HTML5 中移除了。这也就是我只在脚注提到它的原因。请不要将 rel 和它的反义词相混淆。

注 2 在 HTML4 中，不能拥有 id 属性的标签有：base、head、html、meta、script、style 以及 title。而在 HTML5 中任何标签都可以拥有 id 属性。我将这个标签列表重复打印出来，这样你可以明白这很可能不是一个问题。

注 3 在 HTML5 中，所有标签都可以拥有 class 属性。而在 HTML4 中，我前面的脚注中提到的 7 个标签既不能定义 id 属性，也不能定义 class 属性。Param 标签可以定义 id 属性，但是它不能定义 class 属性。重申一遍，这很可能不是一个问题。

```
<div class="vcard">
  <span class="fn">Jennifer Gallegos</span>
  <span class="bday">1987-08-25</span>
</div>
```

单独而言，`<div>` 标签没有任何含义——它仅仅是将其他的标签集合到一起。一个 `` 标签自身也没有任何含义。但如果说，我告诉你 `vcard class` 会将一个人的个人信息整合到一起（此时，不要担心我如何告诉你这些信息；我会在后面讲到）；我告诉你用 `fn class` 标记的标签包含的是一个人的姓名；而用 `bday class` 标记的标签包含的是一个人的 ISO 8601 格式的出生日期，那会怎么样呢？

113

这个 `<div>` 标签立刻就成为对某个人的说明介绍了，它有具体的含义了。现在你也知道“Jennifer Gallegos”是某个人的姓名，而不是某本书的标题了。你也知道“1987-08-25”是某个特定格式的日期，而不是一个恰好看起来像日期的随机字符串了。当你明白 `class` 的某些值的含义的时候，你也理解了那些 HTML 规范并没有定义的应用语义。

在同一文档中的多个标签可以拥有相同的 `class`，而且一个的标签可以拥有多个 `class` 值，这些值之间用空格进行分隔：

```
<ul>
  <li><a class="link external" href="http://www.example.com/>Link 1</a></li>
  <li><a class="link external" href="http://www.example.org/>Link 2</a></li>
  <li><a class="link internal" href="/page2">Link 3</a></li>
</ul>
```

如果你曾经尝试过使用 `id` 属性来作为应用语义的一部分的话，那我现在建议你使用 `class` 属性来进行代替。和 `id` 属性不同，在一个表述中的多个标签可以拥有相同的 `class` 属性。

微格式

我选择了一些相当模糊的 CSS `class` 名称来使得 `<div>` 标签和两个 `` 标签成为某个人的说明描述：“`vcard`”，`fn` 和 `bday`。如果让我自己来制定 `class` 名称的话，我会用类似于 `birthday` 这样描述更准确的名称。但是我没有这样做。我采用了一个已有标准 `hCard` (<http://microformat.org/wiki/hcard>) 中的名称。如果你在某个 HTML 标签中见到了 `class="vcard"` 字样，你就会知道这个标签中的所有内容都应该按照 `hCard` 标准来进行解释。

和 `Maze+XML` 相同的是，`hCard` 标准并不和哪个 RFC 或者互联网草案相关。与 `Maze+XML` 不相同的是，`hCard` 并不是一个个人标准。`hCard` 是一种微格式：一种轻量级的工业标准，它是通过 wiki 上的非正式的协作方式定义的，而不像 RFC 那样要经过

正式的 IETF 流程。

查阅 hCard 标准之后，你就会发现 `fn class` 是用来标记一个人的全名（full name）的，而 `bdai class` 是用来标记一个人的 ISO 8601 格式的出生日期（date of birth）的。现在，当一个文档使用这些 CSS class 的时候，你就知道这个文档的含义了。HTML 标准并没有对于姓名或者生日做任何的说明，但是 hCard 标准却对这些内容进行了讨论。

114 ➤ 微格式使得你可以向 HTML 添加额外的应用语义。HTML 的 `class` 属性，加上 hCard 微格式，就可以让你建立一个能够描述一个人的 HTML 文档。

hCard 仅仅定义了 `class` 属性的值。我所使用的 `` 标签以及 `<div>` 标签，对于 hCard 而言，毫无意义：我可以使用任何其他标签。因为几乎所有的 HTML 标签都支持 `class` 属性，我可以编写一段非结构化的文本，它同时也是个 hCard 文档：

```
<p class="vcard">我的名字是<i class="fn">Jennifer Gallegos</i> 我  
出生于 <date class="bdai">1987-08-25</date>.</p>
```

人类可以将这个表述当作一句话来进行阅读。而 hCard 处理器将忽略所有的“无关的”文字，而将重点放在使用 hCard 的 CSS class 的标签上。

尽管 hCard 微格式没有通过正式的标准化过程，但是它是基于一个已经完成了这些过程的标准而创建的。这个标准就是：vCard，一个在 RFC 6350 中定义的用来展示商业名片的纯文本格式。

我曾在第 5 章中提及过 vCard，并将 vCard 视为一个缺少超媒体控件的领域特定标准的例子。hCard 不过是 HTML 版的 vCard 而已。这也就是 hCard 文档中的最上层的 `class` 的值是 `vcard` 而不是 `hcard` 的原因。

对于哪些类型的信息常常会被添加到商业名片，专家们做了非常多的成本很高的研究调查和长期的讨论，vCard RFC 就是这些研究和讨论的结晶。正如我在第 5 章所说的，我们没有理由仅仅因为 vCard 没有超媒体控件而重复这些研究和讨论。我们可以照抄 vCard 的语义，并将它们应用到一个通用的超媒体语言：HTML 上。

hMaze微格式

在本节中，我将像 hCard 对 vCard 的改造一样对 Maze+XML 进行改造。我会将这个为特定领域（迷宫）设计的、非 HTML 标准改造成一个 HTML 微格式。这样，我就可以使用 HTML 来表示那些基本 HTML 标准所不理解的领域的语义了。

和“hCard”一样，我将我的新的微格式称为“hMaze”（“h”代表“HTML”）。我的微

格式定义了一些特别的 CSS class :

hmaze

表明 hMaze 文档的父标签, 类似于 hCard 的 vcard class。

collection

可能在 hmaze 中出现。用于描述迷宫集合。

maze

可能在 hmaze 中出现。用于描述一个单独的迷宫。

error

可能在 hmaze 中出现。用于描述一条错误消息。

115

cell

可能在 hmaze 中出现。用于描述迷宫中的某个单元格。

title

可能在 cell 中出现。用于包含单元格的名称。

微格式同样可以定义链接关系, 我照抄了 Maze+XML 所定义的全部链接关系。这些关系: north、south、east、west、exit 以及 current 在 class="cell" 的标签中都有着特定的含义 (更确切地说, 是 Maze+XML 标准定义的特定的含义)。当 maze 链接关系出现在 class="collection" 的标签中时, 它也有特殊含义 (就像在 Maze+XML 中那样, 它链接到一个特定的迷宫)。

这就是 hMaze 微格式。最起码是 hMaze 微格式的第一个版本。如果我打算将其发布到微格式 wiki 上, 我就需要编写更多的 CSS class, 比如一些和错误关联的 class, 但是这个作为例子来说已经足够好了。这个微格式可以展示任何 Maze+XML 所能展示的迷宫, 但是它使用的却是 HTML。

我的微格式只定义了 class 和 rel 的值。就像 hCard 一样, 标签的选择权留给了服务器。服务器可以提供或多或少类似于 Maze+XML 的古板的 HTML 文档:

```
<div class="hmaze">
  <div class="cell">
    <div class="title">
      Hall of Pretzels
```

```

</div>
<div>
  <a href="/cells/143" rel="west"/>
  <a href="/cells/145" rel="east"/>
</div>
</div>
</div>

```

或者服务器也可以用一种人类可读的方式来展示相同的数据，这样人类就可以使用他们的 web 浏览器作为 API 客户端了：

```

<div class="hmaze">
  <div class="cell">
    <p><b class="title">Hall of Pretzels</b></p>

    <ul>
      <li><a href="/cells/143" rel="west">Go west</a></li>
      <li><a href="/cells/145" rel="east">Go east</a></li>
    </ul>
  </div>
</div>

```

116 这两个文档都是合法的 hMaze 文档，就 hMaze 而言，它们拥有等价的应用语义。重要的是你要按照 hMaze 规范所要求的那样来使用 class 和 rel 属性（就 HTML 本身而言，这两个文档拥有不同的应用层语义，因为 HTML 的“应用”是人类可读的文档）。

微数据

微数据是针对 HTML5 而对微格式的概念进行的改进。你也看到了，微格式是一种侵入式的。HTML 的 class 属性本来是设计用来传递与可视化显示相关的信息的（通过 CSS），不是用来传递任何应用语义内容的。

HTML 微数据^{注 4}介绍了 5 种新的属性：itemprop、itemscope、itemtype、itemid 和 itemref，它们是专门用来展示应用语义的。这些属性可以出现在任何 HTML 标签中。

我将重点关注它们之中的前 3 个属性。itemprop 属性的用法和微格式中的 class 属性的用法相同。Itemscope 属性是一个 Boolean 属性，用来表明标签中是否包含微数据。Itemtype 属性是一个超媒体控件，它用来告诉客户端从哪里找到这个微数据的含义。

通过稍加改进，微格式的大部分信息都可以展示为微数据。下面的 HTML 文档就展示了一种微数据类型，它是对 hMaze 的轻微的变形体：

```

<div itemscope itemtype="http://www.example.com/microdata/Maze">

```

注 4 一个开放标准，在 W3C 规范中定义，目前是以草案形式出现的。

```

<div itemprop="cell">
  <div itemprop="title">
    Hall of Pretzels
  </div>
  <div>
    <a href="/cells/143" rel="west"/>
    <a href="/cells/145" rel="east"/>
  </div>
</div>
</div>

```

对于微格式而言,客户端仅仅需要知道,如果它发现一个标记为 `class="hMaze"` 的标签,那么这个标签下面的内容就是一个 hMaze 文档。而对于微数据而言, `class="hMaze"` 的标记就不必要了。`itemscope` 属性表示这个标签下面的内容是按照某些文档中规定的规则进行描述的,而 `itemtype` 属性则指向了那个文档^{注 5}。

◀ 117

微格式有一点是优于微数据项的。微数据项不能为 `rel` 属性定义任何值——只能为 `itemprop` 定义。这就表示从严格意义上讲, `rel="east"` 和 `rel="west"` 不是我的那个类似于 hMaze 的微数据项的一部分。即便某个网站 <http://www.example.com/microdata/Maze> 上的文档有可能提到说客户端可以期望在迷宫单元格的表述中看到 `rel="east"` 和 `rel="west"` 这样的内容,但是就微数据标准而言,标准中并没有链接关系的内容。你不能将链接关系定义在一个微数据项里。

微数据项的主要来源是 schema.org, 这是一个由四大搜索引擎公司 (Bing、Google、Yahoo! 和 Yandex) 发起的用于为不同的问题领域定义应用语义的项目。搜索引擎对于理解某张网页的高层应用语义 (也就是网页所讲的关于现实世界的事情) 是很感兴趣的。因为 API 经常处理我们在网页上所讨论的现实世界的事情 (人、产品、事件等), 我们可以在我们的 API 中复用它们的工作。

我将在第 10 章结尾部分罗列出主要的微数据类型, 在那里, 你将会看到一些涉及 schema.org 微数据项的例子。对于人类而言, 这些例子的含义应该是相当明显的。URL <http://schema.org/Person> 指向 schema.org 的数据项, 它与我们日常概念中的 “person” (人) 是一致的。

改变资源状态

我现在已经得到了 hMaze, 并且在某种程度上, 它可以代替 Maze+XML。但是你并没有真正看到我是如何改造 Maze+XML 的。现在让我们添加一个新的特性: 一个神秘的能重新布置迷宫结构的开关。

注 5 这个文档称作 profile, 在第 8 章中, 我将寻找答案来回答它应该是什么样子的。

我将通过定义两个新的 CSS class 来为 hMaze 微格式添加神秘的开关：

switch

可能会在 cell 中出现。描述的是一个开关,这个开关可以设置为两个位置中的一个。每个位置对应于迷宫的一种不同的配置。

position

可能会在 switch 中出现。包含了开关的位置：up 或者 down。

下面是迷宫中一个单元格的表述。你之前见到过,但是现在这个单元格中增加了一个开关：

```
<div class="hmaze">
  <div class="cell">
    <p>
      <b class="title">
        <a href="/cells/H" rel="current">Hall of Pretzels</a>
      </b>
    </p>

    <ul>
      <li><a href="/cells/G" rel="west">Go west</a></li>
      <li><a href="/cells/I" rel="east">Go east</a></li>
    </ul>

    <div class="switch">
      A mysterious switch is mounted on one wall. The
      switch is <span class="position">up</span>.
    </div>

  </div>
</div>
```

当游戏玩家触动开关后,迷宫就完全地改变了自己的配置。如果在客户端触动开关之前,迷宫是如图 7-1 所示的样式,那么之后,它有可能是如图 7-2 所示的样式。

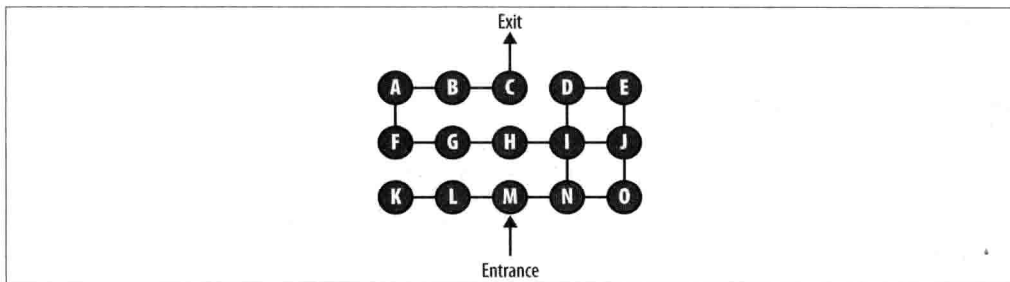


图7-1 触动开关之前

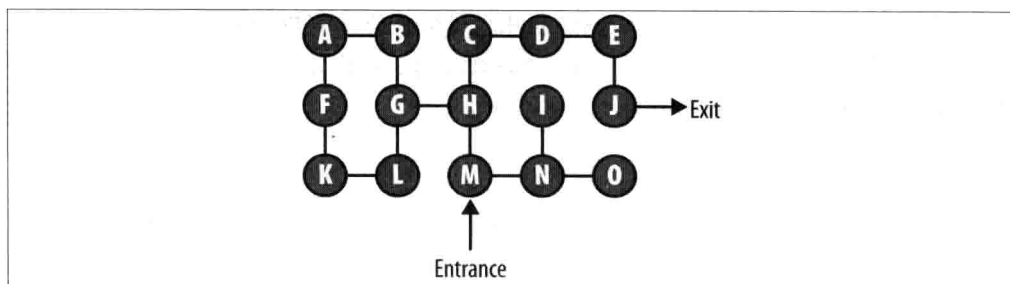


图7-2 触动开关之后

再次触动开关，迷宫就会恢复到最初的状态。

119

很遗憾，由于我还没有解释客户端如何才能真正地触动这个开关，所以你只能相信我的话。我向你保证，这是可行的！我已经设计了一个特殊的 HTTP 请求，客户端能够发起该请求来触动开关。但是我不想告诉你这个请求是什么样子的。你得猜猜看。

好吧。我告诉你。但是我不是用某种语言告诉你——我将使用超媒体。

为表单添加应用语义

hMaze 微格式为 HTML 的 `rel` 属性定义了值，这些值就给 HTML 的链接提供了新的应用语义。当你注意到一个看起来很普通的 HTML 链接拥有一个设置为 `east` 的 `rel` 属性时，这个链接就不再是一个普通的链接了，它就变成了一个穿越地理空间的通道。

`rel` 属性描述了两个资源之间的关系。它说明了一次状态转换：如果客户端访问一个链接，应用状态所发生的变化。

我想要做的是为表示“触动开关”的 `rel` 属性创建一个值。不同于改变应用状态（驱动客户端从迷宫的某个地方到达另一个地方），带有 `rel="flip"` 的链接会改变资源状态。你的应用状态会保持不变——你还是处于标题为 Hall of Pretzels 的单元格中——但是开关可能现在已经是处在另一个状态了，并且迷宫的结构也已经不同了。

这个想法有一个很大的问题。那就是 HTML 链接只支持 GET 方法，而且由于“触动开关”不是一个安全操作，所以我不能使用 GET 方法。“触动开关”会修改资源状态。这就是问题的关键。开关之前的位置是 `off`，现在是 `on`。迷宫的出口曾经在单元格 C 中，现在它在单元格 J 中了。

幸运的是，HTML 同时定义了超媒体表单。一个 HTML 表单可以要求客户端发起一个 POST 请求，而 POST 请求是可以完成任何事情。

对于 HTML，这里还有一个小问题。那就是用来提交 HTML 表单的按钮不支持 rel 属性。但是它们支持 class 和（在 HTML5 中）itemprop。所以让我们定义一个应用到表单提交按钮的 class 属性的应用语义：

flip

可能会在 switch 中的某个表单提交控件中出现。当被激活后，控件将会影响开关的触动。

现在我们已经很清楚如何触动开关了。你首先查看一下 class="switch" 的标签，然后找到里面的 class="flip" 的表单提交控件，最后激活这个控件。再次展示一下那个标题为 Hall of Pretzels 的单元格的表述：

```
<div class="hmaze">
  <div class="cell">
    <p>
      <b class="title">
        <a href="/cells/H" rel="current">Hall of Pretzels</a>
      </b>
    </p>

    <ul>
      <li><a href="/cells/G" rel="east">Go west</a></li>
      <li><a href="/cells/I" rel="west">Go east</a></li>
    </ul>

    <div class="switch">
      A mysterious switch is mounted on one wall. The
      switch is <span class="position">up</span>.

      <form action="/switches/4" method="post">
        <input class="flip" type="submit" value="Flip it!"/>
      </form>
    </div>
  </div>
</div>
```

现在已经清楚了如何触发这个开关了。在那个超媒体表单的指引下，那些理解 hMaze 格式含义的客户端就可以发送一个神奇的 HTTP 请求了，这就是那个我之前曾拒绝说明的这个请求。现在你可以看一下这个请求的内容了：

```
POST /switches/4 HTTP/1.1
Content-Type: application/x-www-form-urlencoded

submit=Flip%20it%21
```

响应可能如下所示：

```
303 See Other
Location: /cells/H
```

客户端会访问 Location 报头中的链接，然后刷新当前迷宫单元格的表述：

```
<div class="hmaze">
  <div class="cell">
    <p>
      <b class="title">
        <a href="/cells/H" rel="current">Hall of Pretzels</a>
      </b>
    </p>

    <ul>
      <li><a href="/cells/G" rel="west">Go west</a></li>
      <li><a href="/cells/C" rel="north">Go north</a></li>
      <li><a href="/cells/M" rel="south">Go south</a></li>
    </ul>

    <div class="switch">
      A mysterious switch is mounted on one wall. The
      switch is <span class="position">down</span>.

      <form action="/switches/4" method="post">
        <input class="flip" type="submit" value="Flip it!"/>
      </form>
    </div>

  </div>
</div>
```

121

资源状态已经改变了！那个“Go west”的链接还在那里，但是指向东边 east 的链接消失了，出现了两个以前并不能访问的新链接（rel="north" 和 rel="south"）。开关还在那里，但是它的位置现在已经不是 up 了，而是 down 了。

虽然我还可以不断地为这个游戏添加新的特性，但是我认为这已经展示了我着手定义一系列的应用语义的过程，这些应用语义可以和那些类似于 HTML 的通用的超媒体格式一起使用。下面将完整的 hMaze 微格式整理到一起。我定义了 7 个 CSS class，它们可以被应用到任何一个标签上：

hmaze

表明 hMaze 文档的父标签，类似于 hCard 的 vcard class。

collection

可能在 `hmaze` 中出现，用于描述迷宫集合。

error

可能在 `hmaze` 中出现，用于表示一条错误消息。

cell

可能在 `hmaze` 中出现，用于描述迷宫中的某个单元格。

title

可能会在 `cell` 中出现，用于包含单元格的名称。

switch

可能会在 `cell` 中出现，描述的是迷宫中出现的开关。

position

可能会在 `switch` 中出现，描述了开关的位置。它有两个位置选项：`up` 或者 `down`。

我定义了 8 个链接关系：`maze`、`start`、`north`、`south`、`east`、`west`、`current` 和 `exit`，它们只能应用到超媒体控件中。这些关系在 `hMaze` 中的含义和在 `Maze+XML` 中的含义是相同的。

122 此外，我还定义了一个只能应用到 `<form>` 标签中的提交按钮的 CSS class（重申，我不得使用 CSS class 是由于 HTML 表单提交按钮不支持 `rel` 属性）：

flip

可能会在 `switch` 中的某个表单提交控件中出现。当被激活后，控件将会影响开关的触动。

这些就是 API 了，更具体地说，这些就是一个 API 规范，一个有着和 `Maze+XML` 一样地位的个人标准了。此外，还有一些我并没有定义的遗留内容：

- HTML 文档究竟应该是什么样子的？我并没有对其进行定义，是因为我对此并不关心。你的 `hMaze` 实现可以提供带有大量背景叙述内容的完整的人类可读的文档，也可以提供针对自动化客户端优化过的非常简洁的文档。只要你正确地使用了 `hMaze` CSS class 和链接关系，你的选择便不会影响迷宫的应用语义。
- 那个神秘的开关可以是带有自己的表述的一级资源吗？例子中的开关看起来好像

拥有自己的 URL (`/switches/4`)，但是客户端从来不会对这个 URL 发起 GET 请求，只会发起 POST 请求。很容易想象一个指向那个 URL 的链接：

```
A <a rel="switch" href="/switches/4">mysterious switch</a>
is mounted on one wall.
```

但是我并没有定义过一个叫作 `switch` 的链接关系，所以这并不是我的设计的一部分。我将会在第 9 章中再回来谈这个想法。

与超媒体相对是普通媒体

我认为将 hMaze 规范与当今所谓的 API 文档做个对比是很有用的。在那种典型的 API 文档中，一连串的服务器端方法被作为独立的 API 调用而暴露出来。每个调用都有它自己对应的 action URL，并且这些调用被极为详细地记录了下来。你以前肯定见到过这种类型的东西。

为了触动开关，请发送一个 POST 请求到：

```
http://api.example.com/switches/{id}?action=flip
{id} 在这里表示对应开关的 ID。
你只有在和开关处在同一个单元格的时候才能触动开关。
```

如果你发现自己正在编写（或者生成）和上面例子类似的文档，你就是在使用人类可读的文档作为超媒体的替代者。这是不可接受的。你正在为你自己和你的用户做无用的工作。

当然，服务器必须通过某种方式提供那些信息。客户端需要了解它究竟要发送什么 HTTP 请求以及在它发送请求之后可能会发生的事情。但是几乎所有的这些信息都可以为潜在的目标读者（软件产品）编写好，并在需要的时候提供出来。你不需要提前用英文将它们拼写出来。

◀ 123

```
<div class="switch">
  A mysterious switch is mounted on one wall. The
  switch is <span class="position">down</span>.

  <form action="/switches/4" method="post">
    <input class="flip" type="submit" value="Flip it!"/>
  </form>
</div>
```

或者，将其他内容去掉只留下最重要的内容：

```
<div class="switch">
  <span class="position">down</span>
```

```
<form action="/switches/4" method="post">
  <input class="flip" type="submit"/>
</form>
</div>
</span>
</div>
```

这个文档对应了协议语义。它说明了客户端究竟要发起什么 HTTP 请求来触发这个状态转换 flip。我仅仅需要提供的人类可读的文档就是 hMaze 规格说明书,它定义了 flip 的应用语义:

flip

可能会在 switch 中的某个表单提交控件中出现。当被激活后,控件将会影响开关的触动。

我并不需要为了构造那个 action URL 而提供一个模板,或者强迫客户端来小心处理“开关的 ID”这样的内部概念,因为用于触动开关的 <form> 标签已经包含了客户端会使用到的那个真正的 URL。我也不需要发出类似于“你只有在和开关处在同一个单元格的时候才能触动开关”的警告,因为超媒体控件只有在能够被使用的时候才能展示出来。如果提交按钮不在那里,状态转换就不可用。

我曾经认为你应该通过标识资源并用超媒体将它们绑在一起的方式来设计 API。如果现在你是以 API 调用列表的方式来发布你的内部方法的,而你又想从中解脱出来,那么面向资源的方法是很好的建议。从资源的角度进行考虑,这至少会使得这些 API 调用能够以一种合理的方式进行分组。

但是在基于超媒体的设计中,资源并不那么重要。设计者的工作是识别出所有的状态转换。一个面向资源的设计方案重点关注的是作为资源的神秘的开关,关注的是它里面的事物。但是开关本身并不是那么重要。我的设计关注的是状态转换,关注的是你能对开关做哪些操作。

HTML的局限性

从技术上说,HTML 是一个领域特定标准,而不是一个通用的超媒体格式。我之所以不在第 5 章而在这里讲解的原因就是 HTML 的“领域”是一个非常通用的领域:人类可读的文档。将 HTML 用于其他目的是毫无问题的,比如由机器人玩耍的迷宫游戏,但是你会很快会遇到这种数据格式的限制。在万维网的世界里,没有人曾经注意过这些限制,但是如果你设计的是一套提供 HTML 的 API,你很快就会注意到这一点。

- HTML 包含了大量的超媒体控件，但是这些控件不能描述所有的 HTTP 协议语义。我们没有办法在不使用 JavaScript 的情况下命令 HTML 客户端去发起一次 PUT 或者 DELETE 请求。
- HTML4 中的表单只能构造两种不同格式的实体消息体：要么是 `application/x-www-form-urlencoded` (基本的键值对)，要么是 `multipart/form-data` (键值对加上文件上传)。
- 不同于 JSON，HTML4 并不能区分字符串和数值。HTML 标签中的任何字符串都被假设为仅仅是——一个字符串。如果你想要说某个字符串应该用一些其他方式来解析，那么，你就需要在 HTML 文档之外自己去定义这些内容。
- HTML4 没有定义展示日期的方式 (JSON 也有同样的问题)。当 vCard 标准定义了 `bdays` 类以后，就意味着，任何提供给 `bdays` 的数据都应该被当作一个 ISO 8601 格式的日期进行解析。没有这些额外的信息 (以一种人类可读的方式展示出来)，我们就没有办法来确定 "1987-08-25" 到底是一个日期还是一个碰巧看起来像日期的字符串。

拯救者HTML5?

新的 HTML5 标准^{注6} 解决了一些 HTML4 中的问题：

- HTML5 定义了时间标签，它可以用来展示某种特定格式的日期或者时间戳。
- 在某些情况下，你可以使用 HTML5 的 `meter` 标签来展示号码，但是通常它并不起作用。
- HTML5 提供了一些新的超媒体控件用于创建内嵌链接，包括：`<audio>`、`<video>`、`<source>` 和 `<embed>`。它们在 API 中都不能使用，除非 API 的部分工作是向人类交付多媒体文件。
- HTML5 定义了一些新的用于验证 `input` 标签的选项。一个 `input` 标签可以指定它想要将日期、号码或者 URL 作为输入。一个 `input` 标签可以被标记为必需的，这意味着不为这个字段提供一个值的话，表单是不能被提交的。HTML5 客户端可以使用这些信息来完成客户端的验证。

在 HTML4 中，验证必须在服务器端完成，或者通过使用自己编写的 JavaScript 代码来完成，这些代码会在客户端试图提交表单的时候执行。

- 我之前提到过 HTML5 定义了微数据属性用于展示应用语义。相较于复用 `class` 属性的微格式，这无疑是一个很大的改进。

注6 开放标准，目前还处于开发阶段。

不幸的是，有一些内容还是没有改变。HTML5 表单仍然不能触发 PUT 或者 DELETE 请求。HTML5 为自己的表单增加了一个新的表述格式：`text/plain`，但是这仅仅是将采用 `application/x-www-form-urlencoded` 格式的键值对转换成相同的纯文本的表述而已。

概括来说，HTML5 提供了一些有用的新特性，但是它没有彻底改变 HTML 作为一种超媒体格式的基础。

超文本应用语言

HTML 是古老的、过于复杂的，并且是为人类可读的文档而设计的。一些新型的超媒体格式、一些设计专门用于 web API 的格式不断涌现出来与 HTML 相映成趣。超文本应用语言（Hypertext Application Language-HAL）就是其中的一种，它采纳了 HTML 的基础性的概念——超链接，其他内容都被无情地砍掉了。我认为它去除的内容过多，但是这不妨碍它成为一个通用超媒体语言的好例子，它不再拥有 HTML 的历史包袱。下面我们看看它是如何工作的。

HAL 有两种形式：一种采用 XML（媒体类型：`application/hal+xml`），另一种采用 JSON（媒体类型：`application/hal+json`）。我将它们分别简称为 HAL+XML 和 HAL+JSON^{注 7}。从形式上说，它们两个是完全相同的。但是我将会把重点放在 HAL+XML 上，因为我认为查看 HAL+XML 文档和了解它的来龙去脉更容易一些。这里是我编写的一个 HAL+XML 文档，一段从假想的 HAL 版本的迷宫游戏获取的表述。它表示迷宫单元格的方式或多或少和 hMaze 相似。它包括一系列的指向其他单元格的链接，以及一个能被触动的开关：

```
<resource href="/cells/H">
  <title>Hall of Pretzels</title>

  <link href="/cells/G" rel="east"/>
  <link href="/cells/I" rel="west"/>

  <resource href="/switches/4">
    <switch>
      <position>up</position>
      <link href="/switches/4" rel="flip" title="Flip the mysterious switch."/>
    </switch>
  </resource>
```

注 7 JSON 版的 HAL 是在互联网草案“draft-kelly-json-hal”中规定的，XML 版本的 HAL 是一个个人标准（http://stateless.co/hal_specification.html）。开发者的计划是为 HAL+JSON 发布一个 RFC，然后紧跟着发布 HAL+XML 的 RFC。

</resource>

HAL 只定义了两个概念：资源和链接。HAL+XML 将它们表示为 <resource> 和 <link> 标签。上面文档中出现的所有其他标签都是我基于 hMaze 而构造的应用特定的标签。

<resource> 标签仅仅表示这个标签内部的 XML 是某些 HTTP 资源的表述。

<link> 标签是一个完全通用的超媒体控件。在超媒体方面，HAL 和 HTML 之间有一个很大的区别。HTML 拥有不同的控件，这些控件分别对应不同的目的。<a> 标签被激活时会发起一个 GET 请求，而当它收到一个文档作为响应后，应用的关注点就会移到那个文档上。 标签会自动发起 GET 请求，在不改变应用的关注点的前提下，将结果表述作为一张图片内嵌到当前文档中。<form> 标签既可以发起 POST 请求也可以发起 GET 请求。但是没有哪个 HTML 控件能够触发 PUT 和 DELETE 请求。如果你想要使用 HTML 描述某个 HTTP 请求，但是 W3C 没有定义相应的标签来完成你想要做的事情的话，你就不走运了。

HAL 只拥有一种超媒体控件，但是这个控件可以做任何事情。它可以触发 GET 请求、POST 请求、带有特定的实体消息体的 PUT 请求。它还可以让用户选择是发送 DELETE 还是 UNLINK 请求。HAL+XML 文档中的 <link> 标签在被激活的时候，可以触发任意的 HTTP 请求。

下面我们就看看 HAL+XML 文档中的链接：

127

```
<link href="/cells/G" rel="east"/>
<link href="/cells/I" rel="west"/>
<link href="/switches/4" rel="flip" title="Flip the mysterious switch."/>
```

这就是我之所以认为 HAL 从 HTML 中去除的内容太多的原因。在这无尽的可能性中，我怎么才能知道所提供的链接中展示的是什么呢？rel="east" 的 <link> 标签应该触发一个 GET 请求，这个请求会给你提供东边的单元格的表述。rel="flip" 的 <link> 标签应该会触发一个 POST 请求来触动开关。它们之中，一个是修改应用状态的安全操作，另一个是会修改资源状态的、不安全的、非幂等的操作。在 HAL 中，这两个链接看起来几乎是完全相同的。唯一真正的区别就只有链接关系了。

HAL 就是通过链接关系来保存那些用于区分各种状态转换的信息的。当我定义了 rel="flip" 的时候，我就是在说这个 flip 状态转换应该通过 POST 请求来触发。这也就意味着编写如下的人类可读的文档：

flip

可能会在 switch 中出现。当通过一个 POST 请求被激活后，控件将会影响开关的触动。

你看到问题了吗？这个 API 的协议语义从机器可读的超媒体中悄无声息地溜进了人类可读的文本中。我们都清楚，命令计算机发起 POST 请求而非 GET 请求是可行的。HTML 标签 `<form action="post">` 就是这个用途。但是 HAL 没有办法用一种机器可读的方式传递协议语义。我不得不将其写进文档中，每个想要实现迷宫客户端的人就必须阅读我的文档，然后将这个协议语义编写进他们的客户端中。

如果一个 API 的应用语义是用文字文档记录的，这是无可厚非的。难的是让计算机理解这些东西（尽管我会在第 8 章中尝试这么做）。但是命令计算机发起一个 HTTP POST 请求却不存在任何困难。

flip 关系是非常简单的，所以可能看起来这不是一个很重要的内容，但是需要记住的是：HAL 中的链接关系可以展示任何一种状态转换，或者状态转换的集合。你可以去了解 HAL 浏览器 (<http://haltalk.herokuapp.com/>)，它是 HAL 的创造者所维护的一个应用程序示例，这样你就会明白我的意思了。

为了在 HAL 浏览器上创建一个账号，你需要激活那个关系为 `ht:signup` 的链接。图 7-3 展示了关于该链接关系的人类可读的文档。

128

signup relation

POST
Create an account

Request
Headers
The request should have the Content-Type application/json

Body
Required properties

- username: string
- password: string

Optional properties

- bio: string
- real_name: string

Example

```
{
  "username": "fred",
  "password": "pwnme",
  "real_name": "Fred Wilson"
}
```

Responses
201 Created
Headers

- Location: URI of the created user account

图 7-3 关于 `ht:signup` 链接关系的全部内容

这是一个很清晰的、有条理的文档，但是它却是那种给 web API 带来恶名的东西。它看起来很像我前面所抨击的“API 文档”。它花费了很多时间来列举触发一个状态转换所必需的 HTTP 请求，仅仅是顺便提到该状态转换的目的是创建一个新的账号。理想情况下，

HTTP 请求和响应应该使用一种机器可读的形式来进行描述，并且只有那些计算机不能理解的内容才会以人类可读的形式保存下来，这就是问题的关键。

HAL 允许通过一个链接关系来触发任何状态转换，但是描述这些状态转换的唯一方式却是编写一系列的人类可读的文章。这不是一种很好的结合方式。

如果你的应用语义与 HTTP 的协议语义之间的差距相对较小的话，这不是什么大问题。对于一个只读的 API，所有的状态转换都是安全的，HAL 也会正常工作。但是你可能需要去看看那个半官方的应用示例就能够了解到 HAL 的限制。在 HTTP 的 POST 方法和 HAL 浏览器的 `ht:signup` 链接关系之间存在着巨大的语义鸿沟。在消除该鸿沟方面，HTML 做的要比 HAL 好得多。

Siren

我将以另外一种通用超媒体格式：Siren^{注8}的简要介绍来结束本章的内容。Siren 是一种比 HAL 还要新的格式，并且，尽管它是基于 JSON 的，但是相对于 HAL 的极简主义，Siren 采纳了更多的类似于 HTML 的方式来实现超媒体。

如下是一个 Siren 文档实例——迷宫单元格的表述，你之前已经见过 HTML 和 HAL 格式的表述了：

```
{
  "class" : ["cell"],
  "properties" : { "title": "Hall of Pretzels" },
  "links" : [

    { "rel" : ["current"], "href" : "/cells/H" },
    { "rel" : ["east"], "href" : "/cells/G" },
    { "rel" : ["west"], "href" : "/cells/I" }
  ],

  "entities" : [
    { "class" : ["switch"],
      "href" : "/switches/4",
      "rel" : ["item"],
      "properties" : { "position" : ["up"] },
      "actions" : [
        { "name" : "flip",
          "href" : "/switches/4",
          "title" : "Flip the mysterious switch.",
          "method": "POST"
        }
      ]
    }
  ]
}
```

注8 个人标准，定义在 Github 的 Siren 页面 (<https://github.com/kevinswiber/siren>)。

```

    ]
  }
]
}

```

Siren 是设计用来展示某种抽象的分组数据的, Siren 将其称为实体 (entity)。从概念上讲, Siren 的“实体”和 HTML 的 `<div>` 标签是很相似的。它是一种可以将你的数据方便地进行分割的方式。一个实体可以是一个拥有自己的 URL 的 HTTP 资源, 但是这并不是必需的。

类似于 Collection+JSON 子项, 一个 Siren 实体定义了一个包含链接的被称为 links 的特殊数据槽。我的单元格实体包含 3 个链接: 指向自己的当前链接、指向其他单元格的 east 和 west 链接。

130 > 这个单元格实体还包含一个子实体: 神秘的开关。这个开关定义了一个称为 flip 的资源状态转换, 我们已经在本章中见到这个状态转换很多次了。

这个 flip 状态转换是通过一个 Siren action: 一个和 HTML 的表单类似的超媒体控件定义的。Siren action 的 name 属性的作用和 HTML 表单的 class 属性以及链接的 rel 属性的作用是相同的。通过这个属性, 我们就可以清楚地了解在客户端决定激活这个控件以后, 哪个状态转换将会发生。触动开关这个状态转换的目的还是需要用人类可读的文字进行描述的。我将这些文字放到了 title 属性中。

这个表述中唯一的新特性就是我开始使用 item 作为一种链接关系:

```

....
"entities": [
  { "class": ["switch"],
    "href" : "/switches/4",
    "rel" : ["item"],
    ...

```

这个链接关系描述的是开关与包含那个开关的单元格之间的关系。Siren 标准要求每个子实体提供一个 rel 属性来描述父实体与其之间的关系。item 链接关系是一个已经在 IANA 注册过的链接关系, 它描述的是某个集合 (单元格) 与它所包含的事物 (比如那个神秘的开关) 之间的关系。

Siren 是处在 HTML 和 Collection+JSON 中间某个位置的一种标准。采用我在第 6 章所讲的集合模式来实现 Siren 这种由嵌套的实体所构成的系统也是可以正常工作的。但是 Collection+JSON 定义了某些种类的资源并且阐明了它们在 HTTP POST、PUT 以及 DELETE 请求下的反应, 而且 Siren 所允许的状态转换要比 HTML 表单所支持的要复杂得多。

Siren 这种方式的好处就是可以更加灵活地展示那些不符合集合模式的状态转换，而缺点就是如果有两个 Siren 应用（或者是两个 HTML 应用），它们之间就不具有那么多的共性了，你需要为此编写比 Collection+JSON 应用更多的特定的客户端代码。

语义挑战：我们现在要怎么做？

让我们再次简要陈述一下所面临的状况。我们拥有一个客户端 - 服务器端的因特网协议，HTTP，它分别为不同的请求：GET、POST、PUT 等各自分配了非常通用的含义。

我们拥有超媒体的概念，它允许服务器端告诉客户端它下一步可能想要发起哪个 HTTP 请求。这样客户端就从需要预先知道 API 的情况中解放出来了。

我们拥有应用语义的概念，它利用某些信息扩展了超媒体控件，这些信息就是当客户端发起某个 HTTP 请求的时候，应用状态或者资源状态所可能发生的特别状况。

131

我们也已经拥有许许多多的用于构建 API 的标准。

我们拥有类似于 Maze+XML 的领域特定标准，它们会为某个小型的问题空间（比如迷宫游戏）定义应用层语义和协议层语义。

我们也拥有类似于 Collection+JSON 和 Atom Publishing Protocol 这样的标准，它们用“集合”和“子项”资源的视角来看待这个世界。这些标准非常详细地定义了协议层语义，但是应用层语义却几乎完全没有定义。一个子项类型的资源必须用某种非常特别的方式响应 HTTP 的 PUT 请求，但是这个子项却完全可以是什么事物。

我们拥有类似于 hCard 的微格式以及类似于 schema.org 的 <http://schema.org/Person> 的微数据项。它们定义了很多应用层语义用来解释某个文档的含义，但是很少或者甚至没有对用来解释 HTTP 请求下相关资源应该如何响应的协议语义进行定义。

我们拥有类似于 HTML、HAL 和 Siren 的标准。这些语言可以让你自由地定义你自己的协议语义以及应用层语义。

我们所面临的挑战是消除我在第 1 章中曾定义的语义鸿沟。提供一套 API，客户端开发人员如何才能编写出一款能够基于该 API 的应用语义进行自主决策的计算机程序呢？

如果这个 API 是通过类似于 Maze+XML 的领域特定标准来进行描述的，消除它的语义鸿沟的办法是很直接的。你所需要的全部信息都已经在那个标准中说明了。它同时解释了协议语义和应用语义。你首先需要阅读这个标准，然后决定你的客户端应该如何响应所有给定的状况，最后编写这个客户端。

但是领域特定的超媒体标准是很少的。大部分的超媒体 API 使用的是类似于 AtomPub 这样的集合标准，或者类似于 HTML 的通用超媒体语言。这些标准定义了 API 的协议语义，但是它们并没有在应用语义上面着太多笔墨。人们必须通过阅读其他文档来理解这些 API 所提供的表述背后所暗藏的含义。

但是那个文档在哪里呢？API 是否需要使用一种微格式呢？使用哪一种呢？你如何找到它呢？你需要了解所有类型的微格式吗？

如果 API 没有使用 HTML，那又该怎么办呢？Siren 并不支持微格式和微数据。如果 API 设计者想要将 hCard 样式的数据添加到 Siren 文档中，那又该怎么办呢？

132 这时候，我们就达到我们当前技术的极限了。对于这些问题，并没有很好的答案。结果就是每个 API 设计者简单地制定一些符合他们已有的服务器端设计的应用层语义，然后将这些语义记录在某个地方。

这就是我们完成 57 个微博 API 的过程。我们现在已经被困住了。API 技术不可能在这些问题尚未被解答前超越“超媒体”状态而进一步发展。在下一章中，我将展示一些初步的答案。

Profile

在过去的 3 章中，我建立了一套用于设计全新的 API 的规则。我们仍需为这些规则做一些工作。但我现在可以通过一些事项来向你呈现这些规则，从而让这些规则更加接近它们的完整形式：

- 针对你的问题是否已经存在某个领域特定的标准？如果存在，便使用它，同时用文档记录下任何应用特定（application-specific）的扩展（第 5 章）。
- 集合模式适用于你的问题吗？如果适用，请选择并采用一个集合标准。然后定义一份应用特定的词汇表并用文档将它记录下来（第 6 章）。
- 如果以上两点都不满足，请选择一种通用的超媒体格式。将你的应用分解成它的各种状态转换，并将这些状态转换用文档记录下来（第 7 章）。
- 到了这一步，你已经明确了你的协议语义，剩下的便都是应用语义的内容了。有没有可以覆盖到你问题领域的现成的微数据项或微格式呢？如果有，请使用它们。否则，请自行定义一份应用特定的词汇表并用文档将它记录下来（第 7 章）。

现在的问题并不是考虑是否要使用“超媒体”。Maze+XML、AtomPub 和 HAL 都是使用超媒体来描述状态转换的，但是它们在使用超媒体时都采用了不同的方式，并用于解决不同的问题。现在的问题是要选择一种格式来让你可以用于表现状态的转换，这些状态转换最终会构建成你的 API。

HAL 非常适合于只读应用；Maze+XML 也非常适合于那些碰巧像迷宫游戏一样的只读应用；而 AtomPub 非常适合于那些工作方式类似于博客的读写应用。如果超出了某个格式的适用范围，你将发现你需要自行去扩展它，不断为它定义更多的扩展，并且为了符合标准所制定的模式从而去定义一些伪资源（fake resources）。

这些规则中的每一条都提到了一件非常重要的事情，而我却尚未讨论过：文档化。“用文档记录下任何应用特定的扩展”，“定义一份应用特定的词汇表并用文档将它记录下来”。那么当我说“用文档记录下来”时意味着什么呢？

134 凭借以往的经验，每当我听闻某人认为需要将 API 文档化时，我脑海中都会充满疑惑。API 社区的社会规范允许通过无数人类可读的文档来弥补人们对于 REST 原则的无知，或是弥补糟糕的设计。我想要减少对人类可读文档的依赖，但是我无法完全摆脱它们。在某种程度上，我必须告诉你在我的迷宫游戏中，`rel="flip"` 表示的意思是触动开关，而非抛掷硬币，也不是在 21 点牌局中翻牌。而超媒体格式本身——HTML、AtomPub 以及其他等的超媒体格式都是定义在像 RFC 这样的人类可读的文档中。

本章的内容将专注于文档化的问题。如果你为这个世界增添了一个新的 API，那么你将真正需要编写的人类可读的文档会有多少呢？这些文档的形式又是怎样的呢？你又该如何避免成为那个创造第 58 个微博 API 的人呢？

客户端如何找寻文档？

在考虑 API 文档应该是什么样子之前，让我们先想想客户端在第一时间是如何找寻文档的呢？Fielding 约束中有一条便是“自描述的消息”。服务器不应该需要去猜测一个 HTTP 请求的意思，而客户端也不应该需要去猜测一个响应的意思。消息本身应该是讲得很清楚的，至少也该是隐含了这些意思。

HTTP 的 `Content-Type` 报头便是这方面最明了的一个例子。这个报头的值告诉了你该如何去解析实体消息体。下面是一些例子：

```
Content-Type: text/html
Content-Type: application/json
Content-Type: application/atom+xml
Content-Type: application/vnd.collection+json
Content-Type: application/vnd.amundsen.maze+xml
```

如果媒体类型定义了超媒体控件（例如 HTML 文档），我们便可以通过解析响应文档从而知道接下来该发送什么样的 HTTP 请求，现在你已经理解了文档的协议语义。如果媒体类型是一种领域特定的格式（比如 Maze+XML），我们也可以通过解析文档，从而对问题空间中的状态（比如一个迷宫单元格）有所理解，现在你又理解了文档的应用语义。一旦你同时理解了协议语义和应用语义，你便完成了。你（或你的软件）已经可以基于可用的信息做出决策了。

大多数时候，你无法只通过媒体类型来同时获得两种语义。考虑下使用了 hCard 微格式的 HTML 文档，我们通过将文档按照 `text/html` 的格式来进行解析，可以得到协议

语义，但是无法得到应用语义。再考虑下你从 Twitter API 获取的 JSON 文档，它是以 `application/json` 格式提供的，你无法通过解析该文档来获取任何的协议语义或应用语义。我们还缺少某些神秘的规范，它们失踪了。

这些“缺失的”规范其实并没有真的丢失。就 hCard 而言，该规范就位于 <http://microformats.org/wiki/hcard>。而对于 Twitter，它的规范位于 <https://dev.twitter.com/docs>。我将这些“缺失的”规范称为 profile，本章的主题就是讨论类似这样的文档。

什么是 Profile?

下面是摘自 RFC 6906 的对于 profile 的正式定义：

定义 Profile 是为了在不改变资源表述本身语义的情况下，使得客户端可以了解到除了由媒体类型定义之外的，与资源表述关联的额外语义……

hCard 微格式显然就很符合这一定义。一个使用了 hCard 的 HTML 文档仍然是一个 HTML 文档，但是它获得了一些大多数 HTML 文档所没有的额外的应用语义。它表述了一个人，虽然没有采用自由流畅的行文，但是却使用了一种让计算机可以编程理解的方式。

Twitter API 的那些人类可读的文档同样也是一种 profile。你可以在没有文档的情况下对一段 Twitter 的表述（仅仅只是 JSON）进行解析，但是你将对它的意义一无所知，它仅仅只是一个 JSON 对象。Twitter 的 API 文档让你可以理解 API 所提供的 JSON 对象的意义（即“使得客户端可以了解到额外的语义”），而不会与 JSON 的规范 RFC 4627 有任何抵触（“不改变资源表述本身的语义”）。

链接到 Profile

使用了 hCard 的 HTML 文档和 Twitter 的 API 所提供的 JSON 文档，在这些表述中所“缺失的”并不是 profile，而是关联 profile 和（使用了该 profile 的）文档之间的连接。我们假设了客户端“已经知道”应该在给定文档上运用哪些 profile。那么，我们知道应该如何解决这个问题，我们可以使用超媒体来为文档建立指向它 profile 的链接。

我们有 3 种不同的方式可以来这样做。让我们来按顺序逐个地看看这些方式。

Profile 链接关系

RFC 6906 定义了一个名为 profile 的链接关系。该关系已经在 IANA 进行了注册，这意味着你可以在任何支持链接关系的超媒体控件中使用 profile：比如由 HTML 定义

的 <a> 标签；由 HTML、HAL 以及 Maze+XML 定义的 <link> 标签；一个 Siren 或 Collection+JSON 的 links 对象；又或者是由 RFC 5988 定义的 HTTP Link 报头。

136 如果你得到了一个以如下形式的内容开头的 HTTP 响应，你便可以知道这是一个使用了 hCard 微格式的 HTML 文档：

```
HTTP/1.1 200 OK
Content-Type: text/html

<html>
<head>
  <link href="http://microformats.org/wiki/hcard" rel="profile">
...
```

JSON 并没有协议语义，也几乎没有应用语义，但是如果你得到了一个以如下形式的内容开头的 HTTP 响应的话，你便可以知道这个文档包含了一个在 JSON 之上的额外的语义层：

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://dev.twitter.com/docs>;rel="profile"

...
```

Profile 媒体类型参数

根据你所使用的媒体类型，你可以通过向媒体类型添加 **profile** 参数，从而在 Content-Type 报头中添加对 profile 的链接。下面是一个 Collection+JSON 文档的 Content-Type 报头的例子：

```
application/collection+json;profile="http://www.example.com/profile"
```

这等于说：“这是一个 Collection+JSON 文档，但是它具有额外的语义，这些语义由位于 <http://www.example.com/profile> 的 profile 进行描述。”

不幸的是，你无法在任意的媒体类型上使用 **profile** 参数。根据 RFC 4288 的 4.3 章节规定，你只能在明确定义了该参数的媒体类型上使用 **profile** 参数。而 JSON 规范并没有提及 **profile** 参数，所以下面这样本让人以为可用的方式其实是非法的：

```
Content-Type: application/json;profile="https://dev.twitter.com/docs"
```

就现在来说，仅有的可以使用 **profile** 参数的超媒体类型只有 Collection+JSON、JSON-LD、HAL 和 XHTML（不是 HTML！）。如果你想要在 HTTP 报头中链接到 profile，而你又没有使用以上的任何一种媒体类型的话，我建议你采用 Link 报头的方式来替代。

特殊用途的超媒体控件

在第7章中我曾展示过 HTML 微数据。我曾说过 `itemtype` 属性是“一个告诉客户端应该去何处查明微数据含义的超媒体控件”。下面是一个例子：

```
<div itemscope itemtype="http://schema.org/Person">
```

我当时并没有这么说，但是它很明显是一个指向 `profile` 的链接。它指向了一个文档，该文档提供了一层凌驾于 HTML 5 规范所定义的内容之上的应用语义。

HTML 4 同样也具有一个特殊的超媒体控件，可用于将整个文档链接到它的 `profile`：

```
<HEAD profile="http://schema.org/Person">
```

```
...
</HEAD>
```

我并不建议你使用这种方式，但是由于历史原因，它还是值得关注的。正如我们将在下一节中所要看到的，这种方式是“`profile`”一词最早的出处。

Profile对协议语义的描述

当 `profile` 在描述 API 的协议语义时，它通常会使用平铺直叙的行文格式。我们可以在那些当今非常流行的 API 的文档中看到这种情况，它们采用普通的行文来描述那些你可以通过发起 GET 请求和 POST 请求而调用的“API 调用 (API calls)”。图 8-1 展示了来自一个 Twitter API 的例子：


<div>  Developers </div> <div> Search API Health Blog Discussions Documentation Sign in </div>	
Tweets Tweets are the atomic building blocks of Twitter, 140-character status updates with additional associated metadata. People tweet for a variety of reasons about a multitude of topics.	
Resource	Description
GET statuses/retweets/:id	Returns up to 100 of the first retweets of a given tweet.
GET statuses/show/:id	Returns a single Tweet, specified by the id parameter. The Tweet's author will also be embedded within the tweet. See Embeddable Timelines, Embeddable Tweets, and GET statuses/oembed for tools to render Tweets according to Display Requirements.
POST statuses/destroy/:id	Destroys the status specified by the required ID parameter. The authenticating user must be the author of the specified status. Returns the destroyed status if successful.
POST statuses/update	Updates the authenticating user's current status, also known as tweeting. To upload an image to accompany the tweet, use POST statuses/update_with_media. For each update attempt, the update text is compared with the authenticating user's recent tweets. Any attempt that would result in duplication...
POST statuses/retweet/:id	Retweets a tweet. Returns the original tweet with retweet details embedded.
POST statuses/update_with_media	Updates the authenticating user's current status and attaches media for upload. In other words, it creates a Tweet with a picture attached. Unlike POST statuses/update, this method expects raw multipart data. Your POST request's Content-Type should be set to multipart/form-data with the media[...]
GET statuses/oembed	Returns information allowing the creation of an embedded representation of a Tweet on third party sites. See the oEmbed specification for information about the response format. While this endpoint allows a bit of customization for the final appearance of the embedded Tweet, be aware that the...

图8-1 一个API调用列表

我们在第 7 章看到过同样的内容，当时我展示了一份用于描述 HAL 表述的人类可读的文档（我们现在可以称它为 profile）：

138

Flip

可能会在 switch 中的某个表单提交控件中出现。当被激活后，控件将会影响开关的触动。

在这两个案例中，我们都看到 API 提供者采用了平铺直叙的行文来描述客户端可以发起的 HTTP 请求。你可以让这些变得更加结构化，甚至让计算机可以理解这些信息，但是它不会是一个 profile：它是超媒体。只有在媒体类型没有超媒体控件（比如 JSON）或者它的超媒体控件还不足以能向客户端准确地说明应该发起哪些 HTTP 请求时（比如 HAL），Profile 才需要对协议语义加以描述。

这就是为什么我推荐你选择一个像 HTML 或 Siren 这样的功能齐全的超媒体格式来作为你的表述格式的原因。虽然如此，你仍然需要编写 profile，但是 profile 不需要包含很多关于你 API 协议语义的细节。这些协议语义将会内嵌在表述本身的内部。

Profile 对应用语义的描述

协议语义处理了 HTTP 请求，但是应用语义涉及到了现实世界中的事物，而计算机在对现实世界的理解方面是很糟糕的。在某些时刻，我们必须要通过编写平铺直叙的行文文档来说明我们的应用语义，从而来消除语义鸿沟。超媒体能够在协议语义方面来拯救我们，但是并没有其他东西能够在应用语义方面拯救我们。

通过这些年来创建的数以千计的 profile，我们可以从中发现一个频频出现的模式。API 的应用语义都趋向于集中在一些短小、神秘的字符串上，比如“fn”、“bday”、“east”和“flip”。

微格式的 profile 是一个包含了那些字符串的列表，每一条都有一段英文说明。而微数据项的 profile 也与此非常相似。传统的 API 文档在协议语义方面花费了很多的时间，但是它同样也在罗列和说明这些神秘的字符串方面花了很多时间。这些字符串可能是 JSON 对象的键，也可能是 XML 标签的名字，又或者是用于扩展 URI 模板或构建查询字符串的变量。

这是一项重大的发现。要让计算机理解名字对人类而言的意义是一项任重道远的任务。但是当我们只是将“fn”做一个 CSS 的 class 时，让计算机明白“fn”只是一个表示某个特殊事物（不管是什么）的神奇的字符串还是相当容易的。

这些神奇的字符串将帮助简化我们的 profile。我将它们分成了两个类别：链接关系和语

义描述符。

链接关系

正如我在第 5 章所说的，链接关系是附属于超媒体控件的一个神奇的字符串，它描述了当客户端触发控件时将可能产生的状态转换。到目前为止你已经在不同的媒体类型中看到过很多链接关系的例子。下面是来自第 5 章的一个 Maze+XML 例子：

139

```
<link rel="east" href="/cells/N"/>
```

下面是一个来自第 6 章的 AtomPub 的例子：

```
<link rel="next" href="/collection/4iz6"/>
```

下面同样是一个来自第 6 章的 Collection+JSON 的例子：

```
{ "name" : "cover", "rel" : "icon", "prompt" : "Book cover",  
  "href" : "/covers/1093149.jpg", "render" : "image" }
```

下面是一个来自第 7 章的 HTML 的例子：

```
<a href="/rooms/154" rel="east">
```

下面是来自第 7 章的一些 Siren 的例子：

```
"links" : [  
  { "rel" : ["current"], "href": "/cells/H" },  
  { "rel" : ["east"], "href": "/cells/G" },  
  { "rel" : ["west"], "href": "/cells/I" }  
]
```

一个支持链接关系的超媒体控件定义了一个用于填写目标 URL 的属性槽（通常称为 href）和另一个用于填写链接关系的属性槽（通常称为 rel）。

就它们本身而言，这些链接关系的名字仅仅是字符串而已：“east”、“next”、“icon”和“current”。人类将会发现这些名字是耐人寻味的，但是在不能确切知道它们是什么意思的时候，我们是没办法通过编程来让计算机理解它们的。这便是 profile 的用途所在了。



作为 API 设计者，你有责任提前在一个 profile 文档或自定义媒体类型的定义中记录下你所有的链接关系。唯一可以例外的是你从 IANA 注册机构选取的链接关系（见第 10 章）。你不能因为觉得这些链接关系是不言自明的，而不承担相应的责任，因为这是你一厢情愿的，它们从来都不可能不言自明。

如果你使用扩展的链接关系（那些看起来像 URL 的链接关系），那么将该 URL 输入 web 浏览器的人将会从该地址得到该链接关系的一个说明。

不安全的链接关系

现在来看看下面这个来自第 7 章的 HAL 的例子：

```
<link href="/switches/4" rel="flip" title="Flip the mysterious switch."/>
```

140 我在另一个例子中曾这样描述过链接——它是一个应用状态与另一个状态之间的迁移，由 GET 请求触发。HAL 的例子比较与众不同，因为它的链接关系（flip）描述了资源状态的变化，而且是由 POST 请求触发的。

Siren action 中的 name 属性也是以相同的方式工作的。它关联了一个神奇的字符串，该字符串潜藏了一个不安全的状态转换。下面是一个与这个 HAL 的例子等价的 Siren 的例子：

```
"actions" : [
  { "name": "flip",
    "href": "/switches/4"
    "title": "Flip the mysterious switch.",
    "method": "POST"
  }
]
```

将这些神奇的字符串称为“链接关系”有一点古怪，因为我们一直认为“链接”是某种由 GET 请求激活的事物。我原本在经过深思熟虑后打算引入一个更加通用的术语“转换关系”，但是事实上一些与 HAL 类似的格式都已经在使用“链接关系”这个词，所以我认为最好还是保持这个惯例，将描述任何状态转换的字符串称为“链接关系”。

语义描述符

现在让我们来聊聊第二种神奇的字符串。hCard 微格式为了标记一个人的全名定义了一个名为 fn 的 CSS class：

```
<span class="fn">Jenny Gallegos</span>
```

下面是一个来自 schema.org 的名为 http://schema.org/Person 的微数据项，它出于相同的目的定义了属性 name：

```
<span itemprop="name">Jenny Gallegos</span>
```

Twitter API 的文档中提到了一个叫作 name 的键，通过将它插入某个 JSON 字典可以用于指明一个与 Twitter 账户相关联的名字（不一定是人类的名字）：

```
{ "name": "Jenny Gallegos"}
```

这 3 种不同的方式都是为了达到相同的目标:指出表述中的哪一部分是属于某人(或某物)的名字。我打算将这一类的事物统称为语义描述符 (semantic descriptor)。

让我们来看看更多的例子。Siren 实体的 class 属性就是一个语义描述符,同样的还有,实体中 properties 的 name 属性:

```
"class" : ["person"],
"properties" : { "name" : "Jenny Gallegos" },
...
}
```

Collection+JSON 子项中的 data 字段的 name 同样也是语义描述符:

```
"data" : [
  { "name" : "family-name", "value" : "Gallegos" }
],
```

在某些特定的 JSON (ad hoc JSON) 中 (不仅仅在 Twitter API 提供的文档中),都会习惯性地使用语义描述符来作为对象的键:

```
{"name" : "Jenny Gallegos"}
```

(我将会在后续的章节中谈到 JSON-LD,它也是基于上述惯例的)

类似地,在某些特定的 XML 文档中,标签的名字通常也相当于语义描述符:

```
<person>
  <name>Jenny Gallegos</name>
</person>
```

但是上述的两种方式仅仅是惯例,客户端通常都不能依赖这些方式。这就是我不认为你应该使用特定的 JSON 或 XML 来设计 API 的一个理由。



作为 API 设计者,你有责任提前在一个 profile 文档或自定义媒体类型的定义中记录下你所有的链接关系。你不能因为觉得这些链接关系是不言自明的,而负起相应的责任,因为这是你一厢情愿的,它们从来都不可能不言自明。

XMDP: 首个机器可读的Profile格式

如果所有你所需要描述的只是这些神奇的字符串、链接关系和语义描述符的话,那么 profile 应该是什么样子的呢?它看上去类似于一个微格式,这便是微格式所干的事。第一个机器可读的 profile 是微格式的描述,这没什么可令人惊讶的。这些 profile 使用了

XMDP 格式，它并不是一种我推荐在如今继续使用的格式，但是由于它足够简单，我们可以通过它来对这个概念做一个很好的介绍。

Profile 的思想最早起源于 HTML 4 规范，该规范提出了一个“元数据 profile”的概念。不幸的是，该规范并没有定义元数据 profile 应该是什么样子的。它仅仅说明了如果你以某种方式获得了一个 profile 的话，应该如何能在 HTML 文档中链接到该 profile。该链接的形式如下所示，我曾在早前向你展示过：

```
142 <HEAD profile="http://example.com/profile">
    ...
</HEAD>
```

由于该规范缺乏对细节上的描述，使得该思想成了摆设。现实生活中并没有元数据 profile 的使用案例（在 web API 之前），也没有关于元数据 profile 应该是什么形式的指导。而将属性置于 <HEAD> 标签中的这种语法也并不优雅。所以 HTML profile 并没有流行起来，而这一思想在 HTML5 中已经被废弃。

但是有一个名叫 Tantek Çelik 的人，他最终让梦想成为了现实。他从 HTML4 规范中搜集了一些模糊的线索，并对正在消失的元数据 profile 标准进行了定义，这就是一种名为 XMDP 的微格式^{注1}。XMDP 是一种用于对其他微格式进行说明的微格式。

用于说明 XMDP 的最好的方式便是去看看它是如何对一种我所讨论过的微格式进行说明的。下面是一个专为 hCard 微格式^{注2}而制定的 XMDP profile 的编辑实例。XMDP profile 罗列了所有 hCard 定义的 CSS class，并给每一个都提供了人类可读的描述信息。而人类可读的描述信息并没有真的告诉你很多的内容。它们只是将你指引到 RFC 2426，该规范定义了基于 hCard 的 vCard 标准：

```
<dl class="profile">

  <dt>class</dt>
  <dd>
    <p>All values are defined according to the semantics defined in the
      <a rel="help start" href="http://microformats.org/wiki/hcard">
        hCard specification</a>
      and thus in
      <a href="http://www.ietf.org/rfc/rfc2426.txt">RFC 2426</a>.</p>
  </dd>

  <dt id="vcard">vcard</dt>
```

注1 XMDP 规范的地址位于 <http://microformats.org/wiki/XMDP>。你可以在该页面（<http://gmpg.org/xmdp/description>）阅读 Celik 对于 HTML4 标准的注释。

注2 关于 hCard 的完整的 XMDP 描述位于 <http://microformats.org/profile/hcard>。你可以将该页面（<http://microformats.org/wiki/hcard>）的内容与 hCard 的人类可读的 profile 版本进行比较。

```

<dd>A container for the rest of the class names defined in this
  XMDP profile. See section 1. of RFC 2426.</dd>
</dt>

<dt id="fn">fn</dt>
<dd>See section 3.1.1 of RFC 2426.</dd>

<dt id="family-name">family-name</dt>
<dd>See "Family Name" in section 3.1.2 of RFC 2426.</dd>

<dt id="given-name">given-name</dt>
<dd>See "Given Name" in section 3.1.2 of RFC 2426.</dd>
...

</dl>
</dd>
</dl>

```

143

这看上去似乎并没太多特殊之处，但是它却带给了我们一个非常有用的功能。计算机可以根据 hCard 的 XMDP 描述来比对 HTML 文档，从而确定哪些 CSS class 是 hCard 的一部分，而哪些不是。

在下面的片段中，**bold** 只是一个普通的 CSS class，这意味着它只是被定义在样式表某处的普通样式。而 **family-name** class 也可以在样式表中定义，但是对于 hCard 来说也同样具有特殊意义：

```
<div class="family-name bold">Gallegos</div>
```

如果脱离了 profile，计算机将没有办法知道“family-name”是 hCard 特定的一种 CSS class。人类必须提前阅读 hCard 标准从而将这方面的知识转化到软件中。一旦有了 XMDP profile，能够理解 XMDP 的计算机通常都可以明白“family-name”对于该 XMDP profile 而言有着特殊的意义。

这并不能给计算机在理解什么是“family-name”方面带来任何帮助，但是这可以让它编写出将 HTML 文档和 XMDP profile 关联起来的简单的软件工具。如果没有 XMDP，你需要为 hCard 专门编写一个工具，并且将需要为 hCalendar、hRecipe 和 XFN 等每个微格式编写同样的工具。

除了 XMDP，Çelik 还定义了一种叫作“rel-profile”^{注3}的简单微格式，该格式仅仅定义了名为“profile”的链接关系：

```
<link rel="profile" href="http://example.com/profiles/microformats/hcard"/>
```

该链接关系最终成为了定义在 RFC6906 中的 profile 链接关系。

注3 更多关于 rel-profile 的信息见 <http://microformats.org/wiki/rel-profile>。

ALPS

XMDP 是个很好的开始，但是我们可以做得更好。我创建了一个叫作 ALPS 的标准，它处理了我在 XMDP 中所遇到的主要问题：XMDP 是被设计用于描述其他 HTML 微格式的一种 HTML 微格式。你只能在表述格式为 HTML 时使用它。

如今，HTML 真的是相当流行。它是在人类 web 领域占据绝对霸主地位的表述格式，但是它并非是 web API 方面首选的格式。这项殊荣应当归属于 JSON，XML 则紧随其后，而 HTML 则只能远远地退居第三、第四的位置。

144

这意味着当今的 API 并没有使用微格式或微数据，即使当使用微格式或者微数据更加有意义的时候。目前并不存在向 JSON 或 XML 文档应用 profile 的规则。我曾看到有 API 设计者一遍遍重复地发明 hCard 和其他微格式的 JSON 版本，而每次的区别都不大。当你使用一种基于 JSON 的表述时，你将无法复用其他人的应用语义，你将需要从头开始编写 profile。当你发布了你的 API 之后，你的用户必须阅读你的 profile 并为实现它的应用语义而编写特殊的代码。

如果有一个像 XMDP 的 profile 格式但并不限于描述 HTML 文档会怎样？如果它可以像在微格式和微数据中发现的那样以一种机器可读的方式表达应用语义，而这种方式又可以转译成 HTML、Collection+JSON、Siren 以及我在本书中所提及的其他超媒体格式，那又会怎样呢？我将尝试把 ALPS 设计成这样一种格式。它虽然不能解决我们所有的问题，但是我认为它代表了我们在这一方向上的进步。

让我们来看看，下面是从一个用于描述 hCard 的应用语义的 ALPS profile 中摘录的片段。其中人类可读的那部分文本是基于原样照抄于 XMDP profile 的，但是文档的结构却大不相同：

```
<?xml version="1.0" ?>
<alps>

  <link rel="self" href="http://alps.io/microformats/hcard" />

  <doc>
    hCard is a simple, open format for publishing people, companies,
    and organizations on the web.
  </doc>

  <descriptor id="vcard" type="semantic">
    <doc>
      A container element for an hCard document. See section 1. of RFC 2426.
    </doc>
    <descriptor href="#fn"/>
  </descriptor>
</alps>
```

```

    <descriptor href="#family-name"/>
    <descriptor href="#given-name"/>
  </descriptor>

  <descriptor id="fn" type="semantic">
    <doc>See section 3.1.1 of RFC 2426.</doc>
  </descriptor>

  <descriptor id="family-name" type="semantic">
    <doc>See "Family Name" in section 3.1.2 of RFC 2426.</doc>
  </descriptor>

  <descriptor id="given-name" type="semantic">
    <doc>See "Given Name" in section 3.1.2 of RFC 2426.</doc>
  </descriptor>
  ...
</alps>

```

该 ALPS 文档为 hCard 微格式定义的每个语义描述符包含了一个 `<descriptor>` 标签。`<descriptor>` 标签的 `type` 属性被设置为 `semantic`，而在 `<descriptor>` 标签中的 `<doc>` 标签包含了人类可读的说明。除了 `<doc>` 标签中的内容，文档中的所有内容都是机器可读的。

从文档的构成上来说，ALPS 相对 XMDP 的主要改进是 `<descriptor>` 标签，它是一个可以将 ALPS 元素链向另一个元素的超媒体链接。`<descriptor href="#fn"/>` 位于 `<descriptor id="vcard">` 中表明了客户端将可以预期看到 `fn` 元素将会嵌套在标记为 `vcard` 的元素之中。在 XMDP 中，信息是以人类可读的文本来传达的，而在此处，它是以机器可读的方式传达的。

hCard 的 XMDP profile 说明了如何以一种 HTML 微格式的方式来表达应用语义。而同一 profile 的 ALPS 版本允许使用微数据来表达那些近似的应用语义：

```

<div itemscope itemtype="http://alps.io/microformats/hcard#vcard">
  My name is <div itemprop="fn">Jennifer Gallegos</div>.
</div>

```

或者作为 HAL 表述的一部分：

```

<vcard>
  <fn>Jennifer Gallegos</fn>
  <family-name>Gallegos</family-name>
</vcard>

```

或者作为 Siren 的实体：

```

{

```

```

    "class": ["vcard"],
    "properties": { "fn": "Jennifer Gallegos" }
  }

```

或者作为某种特定的 JSON:

```

{ "vcard": {
  "fn": "Jennifer Gallegos",
  "family-name": "Gallegos"
}
}

```

下面同样是一段合法的特定的 JSON :

```

{
  "fn": "Jennifer Gallegos",
  "family-name": "Gallegos"
}

```

146 一个 ALPS profile 可以说明所有这些文档的应用语义。你所需要做的全部就是使用 profile 链接关系将文档连接到它的 profile :

但是我们才刚刚开始。ALPS profile 可以像表示语义描述符一样表示链接关系。下面是一个与 Maze+XML 应用语义近似的 ALPS 文档的一个片段 :

```

<?xml version="1.0" ?>
<alps>

  <link rel="self" href="http://alps.io/example/maze" />
  <link rel="help" href="http://amundsen.com/media-types/maze/" />

  <doc format="html">
    <h2>Maze+XML Profile</h2>
    <p>Describes a common profile for implementing Maze+XML.</p>
  </doc>

  ...

  <descriptor id="cell" type="semantic">
    <link rel="help"
      href="http://amundsen.com/media-types/maze/format/#cell-element" />
    <doc>Describes a cell in a maze.</doc>
    <descriptor href="#title"/>
    <descriptor href="http://alps.io/iana/relations#current"/>
    <descriptor href="#start"/>
    <descriptor href="#north"/>
    <descriptor href="#south"/>
    <descriptor href="#east"/>
    <descriptor href="#west"/>
  </descriptor>

```

```

    <descriptor href="#exit"/>
</descriptor>

<descriptor id="title" type="semantic">
  <doc>The name of the cell.</doc>
</descriptor>

<descriptor id="north" type="safe">
  <link rel="help"
    href="http://amundsen.com/media-types/maze/format/#north-rel" />
  <doc>Refers to a resource that is "north" of the current resource.</doc>
</descriptor>

<descriptor id="south" type="safe">
  <link rel="help"
    href="http://amundsen.com/media-types/maze/format/#south-rel" />
  <doc>Refers to a resource that is "south" of the current resource.</doc>
</descriptor>
...
</alps>

```

147

而 cell 描述符是一个语义描述符，就好比 hCard 的 ALPS profile 中的 vcard 元素。但是 north 和 south 描述符表示了链接关系。它们将自己 type 设置为 safe，这意味着 north 和 south 是可以由 HTTP GET 触发的安全的状态转换。

如果你使用 HTML 微数据来表达这些语义，它看上去便是如下的形式：

```

<p itemtype="http://alps.io/example/maze#cell">
  You are in the <span itemprop="title">Foyer of Horrors</span>.
  Exits: <a href="/cells/I" rel="north">north</a>,
        <a href="/cells/M" rel="west">west</a>,
        <a href="/cells/O" rel="east">east</a>.
</p>

```

如果你使用 Siren 文档来表达类似的语义，它看上去将会是这样的：

```

{
  "class": ["cell"],
  "properties": { "title": "Foyer of Horrors" },
  "links": { "north": "/cells/I",
             "west": "/cells/M",
             "east": "/cells/O" }
}

```

ALPS 同样也可以用来描述不安全的状态转换。下面是从一个用于描述 hMaze 应用语义的 ALPS 文档中节选的一段片段（下面是一个针对开关的语义描述符，以及一个针对触

动开关的非安全的状态转换)：

```
<descriptor id="switch" type="semantic">
  <doc>A mysterious switch found in the maze.</doc>
  <contains href="#flip"/>
</descriptor>

<element id="flip" type="unsafe">
  <description>Flips a switch.</description>
</element>
```

此处的 `type` 是 `unsafe`，表明该状态转换既不是安全的也不是幂等的。ALPS 同样也定义了 `type="idempotent"`，用于描述幂等但不安全的迁移。

ALPS 并不说明应该使用什么 HTTP 方法来发起不安全或幂等的状态转换，这方面的决策权留给了超媒体控件。ALPS 只是说明了当超媒体控件被触发后将会发生的状态转换。在 HTML 中，该 HTTP 方法将会是 POST：

```
<form action="/switches/4" method="POST">
  <input type="submit" class="flip" value="Flip it!">
</form>
```

148 而在 HAL 文档中将完全不会指定 HTTP 方法：

```
<link href="/switches/4" rel="flip"/>
```

ALPS的优势

HTML 文档可以通过使用 `profile` 链接关系来链接到 ALPS profile 从而调用该 profile。下面是一个通过 `profile` 链接关系引入了 ALPS profile 的 HTML 文档，该 profile 与 hCard 的应用语义近似：

```
<html>
  <head>
    <link href="http://alps.io/microformats/hcard" rel="profile"/>
  </head>

  <body>

    <p>Some unrelated content.</p>
    <div class="vcard">
      <span class="fn">Jennifer Gallegos</span>
      <date class="bday">1987-08-25</span>
    </div>

    <p>More unrelated content.</p>
```

```
</body>
</html>
```

该页面的 body 部分只有 HTML 加上 hCard。不管怎样，考虑到这可能是一个能够理解 HTML 和 ALPS，但是并不理解 hCard 的客户端。该客户端可以下载 ALPS profile，从而通过使用该 profile 在 HTML 中准确地识别出 hCard 文档。客户端可以定位像 fn 这样独立的数据位，并依靠其人类可读的描述来交叉引用这些数据——尽管还是没有办法来理解 hCard 的意义。

而且我们并不仅仅限于 HTML，你已经看到 ALPS profile 可以被 Siren、HAL、XML 以及 JSON 等表述所使用。它同样也可以和 Collection+JSON 一起工作：

```
{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://www.example.com/jennifer",

    "links" : [
      { "href" : "http://alps.io/microformats/hcard",
        "rel": "profile"
      }
    ],

    "items" : [
      { "_class" : "vcard",
        "fn" : "Jennifer Gallegos",
        "bday" : "1987-08-25"
      }
    ]
  }
}
```

149

我们并不需要创建一个特殊的 jCard 微格式^{注4}（通过向 JSON 增加 hCard 的应用语义而产生的格式）。你只需要使用 hCard 的 ALPS profile 就可以了。

一个 ALPS 文档可以将 hCard 的应用语义带给很多不同的超媒体格式，并赋予那些特定的 XML 和 JSON 文档以超媒体能力。所有你需要做的就是使用 profile 链接关系来将表述链接到 ALPS 文档。

现在我们没有理由来一遍又一遍地重新发明微格式和微数据项。每当一个新的超媒体格式登场时，定义好如何将 ALPS profile 应用于这种新的格式将使得它可以适用于每一个

注4 这看上去貌似是个愚蠢的主意，但是也存在着一个做了非常相似的事情的互联网草案（draft-ietf-jcardcal-jcard）。同样还有一个定义在 RFC 6351 的 xCard 标准，它是 vCard 向 XML 的简单移植。

使用这种新格式的文档。

即使你要坚持使用 HTML（可以天然地使用微格式和微数据），你也可以从 ALPS 文档中找到有用的部分。我在 <http://alps.io/> 建立了一个可搜索的 ALPS 文档的仓库。我将在第 10 章的“语义公园”一节中讨论更多关于这个仓库的内容。该仓库囊括了大部分微格式、schema.org 上大部分微数据项以及一些其他标准的 ALPS 版本。

ALPS 仓库可以帮助你非常容易地查找和复用那些别人已经定义好的独立的语义描述符和链接关系——即使它们所定义的元素所采用的表述格式并非是你所使用的表述格式。通过将你的 ALPS 文档上传至仓库，你就可以让其他对该文档感兴趣的组织复用你的成果，而无须通过正式的标准流程。

你可以先编写 ALPS 文档，然后采用简单的 XSLT 样式表将它转换成人类可读的文档，从而代替原来直接编写人类可读文档的方式。ALPS 同样也允许以新的方式来将表述集成到开发工具。想象有这么一个 IDE 插件，它可以找到文档中全部的链接关系和语义描述符，并且当鼠标悬停在这些元素之上时说明这些内容的含义。回到每个微格式还定义在它们各自的人类可读的文档中的年代，编写这样的工具将需要开发者理解每个单独的微格式，并且当有新的微格式被批准后需要承诺添加对该格式的支持。而现在，你在编写工具时只需要理解 ALPS 而无须基于其他任何的条件。

150 我并不是说任何人都将编写这样的工具，但是它阐明了一种模式。一个理解 ALPS 的工具无须一次一个地为 profile 添加支持。通过支持 ALPS，你将支持所有微格式的 ALPS profile 和 schema.org 上全部的微数据项的 ALPS profile，以及更多的 ALPS profile。

ALPS 还有一些功能我将不会涉及到，因为我不想将我的宠物标准写成一部书。我将在本书的后续内容中贯穿始终地使用 ALPS 的片段，以此作为采用机器可读形式来表达应用语义的速记方式。如果你对将 ALPS 作为标准感兴趣的话，请访问 ALPS 的网站 <http://alps.io/>。正如我所写到的，我正在为制定 ALPS 的规范工作着，并且计划将它作为一项互联网草案进行提交。

ALPS并不是万金油

和 XMDP 类似，ALPS 并不具备当你在编写机器可读的 profile 时可能想要的所有功能。我为了保持 ALPS 的简单、灵活以及跨媒体类型的通用性，从而省略了很多功能。

ALPS 一种非常宽容的格式。它为 API 中那些神奇的字符串提供了人类可读的定义，并为如何在表述中找到这些字符串提供了粗略的指引。它并不会以机器可读的方式来说明某些事情，比如某个语义描述符是必需的，又或者该语义描述符在某一地方只能出现一次。

假设你定义了一种 HTML 微格式，它指明文档中具有特定 id 属性的标签将具有特殊的意义：

```
<div id="a-very-important-tag">
```

在第 7 章中，我曾反对过这么做，但是有时它还是会出现。在 ALPS 中我们没有办法来表示该特殊的 id。我曾在 ALPS 中省略了这项能力，因为只有基于 XML 的表述支持在单个文档范围内可以拥有唯一的 ID。你可以使用人类可读的文本来说明某个给定的语义描述符在每个文档中只能被使用一次，但是你无法使用 ALPS 来以机器可读的方式进行说明。

在第 12 章中，我将会谈到 RDF Schema，一种比 ALPS 发展得更加深远的 profile 语言，它可以使应用语义变得机器可读。我之所以还会设计 ALPS 是因为我发现对于大部分想要使用它的开发者来说 RDF Schema 过于复杂了。

JSON-LD

JSON-LD^{注 5} 是另一种由于人们不堪使用 RDF Schema 而发明的 profile 语言。JSON-LD 让你可以将一个被称为上下文(context)的机器可读的文档与普通的 JSON 文档结合起来。这让我们可以很容易地为一个现有的 API 定义 profile，而无须更改文档的格式，从而避免破坏现有的客户端。

◀ 151

JSON-LD 源自 RDF 传统的方法，当我在第 12 章讨论完 RDF 之后，我会重新回来谈这一点。但是你无须对 RDF 有任何了解便可以将 JSON-LD 作为一种简单的 profile 语言来使用。

下面是一段特别有代表性的由当今的许多 API 所提供的未加修饰的 JSON 表述：

```
HTTP/1.1 200 OK
Content-Type: application/json

{ "n": "Jenny Gallegos",
  "photo_link": "http://api.example.com/img/omjennyg" }
```

通过人眼的视角来观察它，我们有一段数据（一个字符串）拥有一个语义描述符（n）以及一个超媒体链接，它的链接关系具有一个笨拙的名字叫作 photo_link。如果以一个自动化客户端的视角来观察它，我们将什么也看不到。因为 application/json 媒体类型并没有超媒体控件，那个链接仅仅是一个正好看上去很像 URL 的字符串。字符串“n”并不是语义描述符，它只是一个字符串。而字符串“photo_link”也不是一个链接关系，

注 5 一个还在制定中的开发标准，定义见 <http://json-ld.org/spec/latest/json-ld/>。

它也只是一个字符串。

像这样的 API 成百上千。事实上，这是我在编写本书时的一个现状。我尝试给这些新的 API 的设计者们提供工具，从而帮助他们做得更好，但是那些现有的 API 怎么办？我们能否在不改变文档格式的情况下改进这些 API 呢？

我们可以通过将 API 所提供的每个文档链接到它的人类可读的 profile（即它的 API 文档）来稍微改善这一状况：

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <http://help.example.com/api/>;rel="profile"

...
```

但是这并不能带来太多的帮助。

下面我们来看看 JSON-LD 将会怎么来处理这些问题。我们将会提供一个链向 JSON-LD 上下文的链接，从而取代原来链向人类可读的 profile 或 ALPS profile 的链接。此处的链接关系比在 IANA 注册的关系 profile 更为特殊，它是一个被特别设计用来链向 JSON-LD 上下文的扩展关系：

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <http://api.example.com/person.jsonld>;rel="http://www.w3.org/ns/json-ld-#context"

{ "n": "Jenny Gallegos",
  "photo_link": "http://www.example.com/img/omjennyg" }
```

152 > 通过向 `http://api.example.com/person.jsonld` 发起第二个 HTTP GET 请求，你将会找到对应的上下文。而 HTTP 响应看上去将会是如下的形式：

```
HTTP/1.1 200 OK
Content-Type: application/ld+json

{
  "@context":
  {
    "n": "http://api.example.org/docs/Person#name",

    "photo_link":
    {
      "@id": "http://api.example.org/docs/Person#photo_link",
      "@type": "@id"
    }
  }
}
```

```
}  
}
```

任何定义了属性 `@context` 的 JSON 对象都可以是 JSON-LD 上下文。这个特殊的上下文将依据人类可读的 API 文档对 JSON 表述的应用语义进行说明。

其中 `n` 的值是一个 JSON 字符串，而 JSON-LD 将会将它当成一个 URL 来进行解释。不管 URL 的背后是什么内容，这些内容都将对最初的 JSON 文档中的 `n` 属性的应用语义进行说明。

这就是说……

```
"n": "http://api.example.org/docs/Person#name"
```

……如果你对如何理解下面的内容感到困惑的话……

```
"n": "Jenny Gallegos"
```

……你可以访问 <http://api.example.org/docs/Person#name> 并阅读相应的说明。与 ALPS profile 不同，JSON-LD 上下文通常不会直接对应用语义进行说明，它们采用链接来指向位于别处的一个说明。

而 `photo_link` 的值是一个 JSON 对象：

```
"photo_link":  
{  
  "@id": "http://api.example.org/docs/Person#photo_link",  
  "@type": "@id"  
}
```

在 JSON-LD 中，`@id` 通常意味着“超媒体链接”。而对象的 `@id` 属性是一个链向该词条的应用语义说明的链接。

该词条的 `@type` 属性同样也是 `@id`。这就是将 JSON 文档转化成超媒体文档的魔法。将 `photo_link` 的 `@type` 设置为 `@id` 表明了 JSON 文档中的 `photo_link` 无论在何时出现，客户端都可以将它作为一个超媒体链接来处理，而并非当作一个只是看上去像 URL 的字符串。

153

多亏了 JSON-LD，我们的首个原本让人绝望的 HTTP 响应，变成了一段自描述的消息：

```
HTTP/1.1 200 OK  
Content-Type: application/json  
Link: <http://api.example.com/person.jsonld>;rel=  
      "http://www.w3.org/ns/json-ld#context"
```

```
{ "n": "Jenny Gallegos",
  "photo_link": "http://www.example.com/img/omjennyg" }
```

计算机将可以通过将 JSON 文档和 JSON-LD 上下文结合起来从而识别出超媒体链接。在本例中，这几乎就是计算机所能做的全部工作了。我们的 JSON-LD 上下文包含了链接到资源类型 `n` 和 `photo_link` 的描述，但是它们的链接最终都链接到了现存的人类可读的 API 文档。

但是 JSON-LD 上下文可以简单地链接到关于应用语义的一段机器可读的 ALPS 描述：

```
{
  "@context":
  {
    "@type": "http://alps.io/schema.org/Person",
    "n": "http://alps.io/schema.org/Person#name",
    "photo_link":
    {
      "@id": "http://alps.io/schema.org/Person#image",
      "@type": "@id"
    }
  }
}
```

或者，JSON-LD 上下文也可以如我们将在第 12 章中所看到的那样，根据诸如 FOAF 这样的 RDF 词汇表来描述资源的应用语义：

```
{
  "@context":
  {
    "@type": "http://xmlns.com/foaf/0.1/Person",
    "n": "http://xmlns.com/foaf/0.1/name",
    "photo_link":
    {
      "@id": "http://xmlns.com/foaf/0.1/image",
      "@type": "@id"
    }
  }
}
```

154 不管你怎么做，目标都是一样的。JSON-LD 让你通过在一个普通的 JSON 文档之上添加上下文，从而可以对该文档的应用语义进行说明。

内嵌的文档

下面是来自第 7 章迷宫游戏中的用于切换神秘开关的 HTML 表单：

```
<form class="flip" action="/switches/4">
  <input type="submit" value="Flip it!"/>
</form>
```

下面是一个 HAL 的版本：

```
<link href="/switches/4" rel="flip" title="Flip the mysterious switch."/>
```

下面是一个 Siren 版本：

```
"actions" : [
  { "name": "flip",
    "href": "/switches/4"
    "title": "Flip the mysterious switch.",
    "method": "POST"
  }
]
```

HTML button 的 value、HAL link 的 title 以及 Siren action 的 title 全都是用于对超媒体控件进行说明的人类可读的文本。对于表单的字段来说，HTML 的 label 标签和 Collection+JSON 的 prompt 属性所起到的用途是相同的。在所有这 3 个例子里，原本你可能期望应该在一个独立的 profile 中找到的应用语义的记录，目前都内嵌在文档自身内部。

这里有一件不可思议的事情：这些文本是多余的。“Flip the mysterious switch.” 并不是一个 profile，因为没有任何东西专门将英文文本和链接关系 flip 关联起来。如果你理解这个链接关系，那么人类可读的文档对你来说将没什么作用；又或者你不理解这个链接关系，那么这个链接关系将没有意义。为什么一个文档应该要包含同一应用语义的两种表述：人类可读的表述和机器可读的表述呢？

这两种不同版本的语义面向了不同的目标受众。这种冗余让人类驱动的客户端与自动化的客户端都可以使用相同的表述。flip 关系的正式定义（由 profile 所展示的）是为客户端程序员而编写的，而英文文本（内嵌在文档内部）主要是供人类消费的。

一个人类用户将会忽略掉 profile。人类用户会阅读到“Flip the mysterious switch.”的提示，然后决定是否要切换开关。而一个自动化的客户端将会忽略掉内嵌的文档，它理解了 class="flip"（或者是 rel="flip" 又或者是 "name": "flip"）的含义之后，会将超媒体控件与 profile 中所展示的 flip 的意义关联起来，并以这些内容为基础做出决策。

◀ 155

如果你正在设计一个 API，而你确信所有状态转换的决策都将由人类作为最终用户来做出的话，那你就完全不需要 profile，比如各类网站就没有 profile。如果你确信所有决策将需要由自动化的客户端来做出，那么你将完全不需要内嵌的文档。

但是在现实中,你不可能对此一清二楚。你很可能不知道你的用户会用你的 API 做些什么,而你必定也不会知道将来会发生些什么。最好的策略就是定义一个 profile 以供编写自动化客户端时之用,同时又在你的表述中内嵌了自然语言的文档,以周全人类最终用户的利益(当然,假设媒体类型支持它)。

标记 `title="Flip the mysterious switch"` 并没有对标记 `rel="flip"` 进行说明。它们是面向了不同的受众对同一件事的两种说法。内嵌的文档可以是有价值的,但是它是通过引入的昂贵硬件——人类来“解决”语义挑战的。最好是在你已经知道由人类来做决策的情况下才保留这种方式。

总结

本章的内容覆盖的范围和领域特别广,我不得不定义一种全新的数据格式来讲述我需要讲述的这个故事,但是我们现在终于就要结束这趟开始于第 5 章的旅行了。我们可以通过结合精心选择的媒体类型和 profile 来跨越鸿沟从而解决语义挑战的问题。下面是用以解决该问题所需的一些基本信息:

- 链接关系是一个字符串,它描述了当客户端触发超媒体控件时将会发生的状态转换。举例说明:Maze+XML 中的 `east` 关系,通过这个关系,你就可以知道某个确定的链接是指向当前资源的地理上的东边方向的某个事物。从习惯上来说,状态转换是应用状态的一次变化(由 GET 请求触发),但是它也可以是一次资源状态的变化(由 PUT、POST、DELETE 或 PATCH 触发)。
- 语义描述符是一个简短的字符串,它用以表明表述中某些部分的意义。举例说明:hCard 的 `fn` 描述符,它在 HTML 中是作为 CSS class 来标记人的姓名的。它不像“链接关系”,这是我为本书制造的一个词。
- 虽然链接关系和语义描述符本身都没有意义,但是通常它们附近都会有一些包含了人类可读的说明的文档。我们将这些文档称为 *profile*。
- Profile 历来都采用单调乏味的“API 文档”的形式。但是如果你为你的表述选好了一个不错的媒体格式,你的 profile 将只是一个由链接关系和语义描述符组成的列表,每一个条目都配有平铺直叙的文本加以说明。这一优化让你可以使用 XMDP、ALPS 或 JSON-LD 来创建机器可读的 profile。
- 机器可读的 profile 允许客户端自动查找链接关系或语义描述符的(人类可读的)定义。机器可读的 profile 可以被搜索和混合。ALPS 注册表(<http://alps.io>)包含了大量可以用以工作的 ALPS profile。
- JSON-LD 上下文可以接受当今 API 所提供的特定的 JSON 文档,并以机器可读的方式来描述它们的应用语义和协议语义。你可以在不破坏现有 API 客户端的情况

下，使用 JSON-LD 来改进 JSON API，让它具有简单的超媒体控件。

- ALPS profile 是一种与表述类型无关的 profile。一个 ALPS profile 可以应用于 HTML 文档、HAL 文档、Collection+JSON 文档、特定的 JSON 或 XML 文档，以及更多其他的类型。
- Profile 并不是内嵌于超媒体表述中的人类可读文档的替代，这是两种不同的用例。Profile 使得开发者可以编写智能的客户端，而表述中的内嵌文本使得人类可以通过能忠实地呈现表述的客户端来使用某个应用。
- 在下一章中，我将会介绍一种用于设计（基于超媒体的）API 的通过程，并以此来对过去几章的内容进行总结。

API设计流程

这个流程会花费很多的时间，但是现在我能够解决那些曾经促使你来购买这本书的基本问题了。你需要设计一个 API：它应该是什么样子的呢？在本章中，我会提出一个设计流程，这个流程以业务需求为起点，以一些软件和人类可读的文档资料为终点。

两个步骤的设计流程

在这种最简单的形式中，流程只包含两个步骤：

1. 选择要在你的表述中使用的媒体类型。这一媒体类型就限定了你的协议语义（在 HTTP 协议下你的 API 的反应）以及你的应用语义（你的表述所涉及到的在现实世界中的事物）。
2. 编写包含其他所有内容的 profile。

这种流程并不会必然为你带来一个良好的 API。事实上，每个 API 都曾经设计过这样的流程。如果你希望得到一个真正通用以至于都难以学习的设计，那么你尽可以在第一步中选择 `application/json` 作为你的表述格式。因为 JSON 没有对你的协议语义和应用语义施加任何的约束条件，你就需要将大部分的时间花费在第 2 步中：定义一个 fiat 标准并用人类可读的 API 资料来进行描述说明。

当今的大部分 API 都是这样设计的，这也是我试图阻止的设计流程。大量的用于创建一种 fiat 标准的工作是没有必要的，基于 fiat 标准编写的客户端代码也是不能被复用的。在做任何其他的工作都必须要做一些前期的思想准备并有愿意在可能的情况下复用他人的工作。

七步骤设计流程

所以我将上述流程扩展为 7 个更加细化的步骤。提前做一些准备性工作会有助于你选择一种表述格式并尽可能地保持你的 profile 的简单。

1. 将客户端可能想要从你的 API 中取出或者放进你的 API 中所有信息都罗列出来。这些信息将会变成你的语义描述符。

语义描述符常常会组成一种层级结构。一个指向的是现实世界中某个对象（比如一个人）的描述符通常会包含许多的更加细化、更加抽象的描述符（比如姓名（givenName））。你需要将你的描述符以一种直观的方式进行分组。

2. 为你的 API 画一幅状态图。图上的每个方框都代表一种表述——一个将你的某些语义描述符组合到一起的文档。你应该使用箭头来将表述以某种方式连接起来，你的客户端应该能够通过这种方式来找到这些箭头的本质含义：它们就是由 HTTP 请求触发的状态转换。

你还不需要为你的状态转换分配具体的 HTTP 方法，但是你应该清楚每个状态转换的情况：它们是安全的，还是不安全但是幂等的，又或者是既不安全也不幂等的。

在这个时候，你可能会发现某些你之前还认为是语义描述符的事物（某项业务的客户）作为一种链接关系（某项业务使用链接关系 customer 来链接到某个人或者另一项业务）来处理更加有意义。重复步骤 1 和步骤 2 直到你对你的语义描述符和链接关系已经满意。

现在你已经理解了你的 API 的协议语义（客户端将来会发起哪种 HTTP 请求）和应用语义（哪些数据会在客户端与服务器之间进行发送和接收）。你已经提出一些充满魔力的字符串（语义描述符和链接关系），就是这些字符串使得你的 API 成为了独一无二的 API。你也大概知道了这些字符串如何将如何融入到 HTTP 的请求和响应中。你之后就可以执行下面的步骤了。

3. 调整你的那些充满魔力的字符串，使它们与已有的 profile 保持一致。我在“语义动物园”一章中列举了一些用于查看那些信息的地方。值得考虑的有 IANA 上注册的链接关系、schema.org 或者 alps.io 提供的语义描述符、领域特定的媒体类型所提供的名字，等等。

这可能改变你的协议语义！尤其是，不安全的链接关系可能在幂等或者非幂等的模式间来回切换。

重复第一步到第三步，直到你对你的那些字符串的名称以及状态图的布局已经满意。

4. 你现在该准备选择一种媒体类型（或者定义一种新的媒体类型）了。媒体类型必须和你的协议语义以及应用语义相兼容。

如果你足够幸运，你可能会找到一种已经覆盖你的部分应用语义的领域特定的媒

体类型。如果你是从头定义一种你自己的媒体类型的话，你可以让它完全符合你的需要。

如果你选择了一种领域特定的媒体类型，你可能需要回到第 3 步，对你的语义描述符和链接关系的名称进行调整以和该媒体类型所定义的名称保持一致。

5. 编写一份记录你的应用语义的 profile。这个 profile 应该说明了你所有的充满魔力的字符串，除了那些 IANA 注册的链接关系以及该媒体类型已经解释过的字符串。我建议你将 profile 编写为一份 ALPS 文档，但是 JSON-LD 上下文的文档或者一个普通的网页也都是可以的。你在第 4 步中从其他人那里借鉴的语义越多，你在这一步所必须完成的工作就越少。

如果你自己定义媒体类型的话，你或许能够跳过这一步，这依赖于你将多少信息记录到媒体类型规范说明书中。

6. 现在可以开始编写代码了。开发一款实现了第 3 步中的状态图的 HTTP 服务器。客户端在发送某个 HTTP 请求之后应该触发恰当的状态转换进而收到某个表述作为响应。

每个表述都会采用你在第 4 步中所选择的媒体类型，并链接到你在第 5 步中定义的 profile。它的数据载荷(data payload)会为你在第 1 步中定义的语义描述符赋值。它还会包含超媒体控件用于告知客户端如何触发你在第 2 步中定义的更多的状态转换。

7. 发布你的告示牌 URL。如果你已经正确地完成了前面 5 个步骤，这个 URL 就是你的用户开始使用你的 API 所唯一需要知道的信息。你也可以编写一些备用的人类可读的 profile (API 资料)、教程以及有助于你的用户入门的客户端示例，但是这些都不是设计的一部分。

现在让我们采用第 7 章中的迷宫游戏作为示例来进一步地了解每个步骤。

第1步：罗列语义描述符

如下是迷宫游戏中所有起作用的数据：

- A maze
- A maze cell
- A switch
- The position of a switch (“up” or “down”)
- The title of a maze cell
- A doorway connecting one maze cell to another
- An exit from a maze
- A list of mazes

当我试图将它们放到一个层级结构中时，如下是我找到的内容。

- A list of mazes
 - A maze
 - A maze cell
 - A title
 - A doorway connecting one maze cell to another
 - An exit from the current maze
 - A switch
 - A position (“up” or “down”)

图 9-1 展示了我为了将数据分成表述所做的第一次努力。我得到了语义描述符的层级列表，并用方框将我认为属于一起的数据块圈了起来。

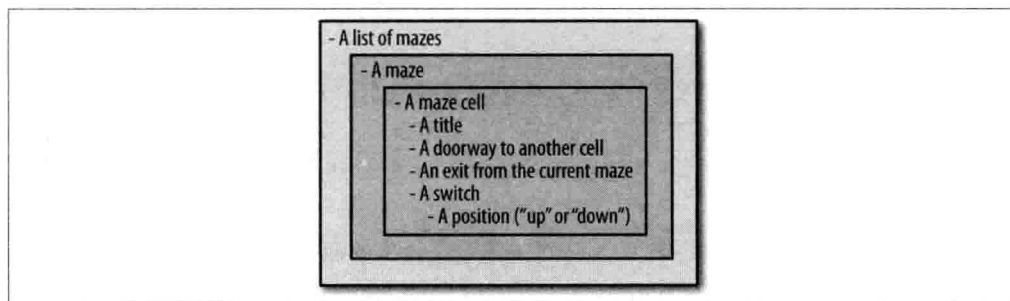


图9-1 将描述符分成表述

这些分割的数据为我提供了 3 种不同的表述：迷宫列表、单个迷宫、迷宫中的单元格（可能包含或不包含开关）。

161 第2步：画状态图

现在的问题是：这些表述是如何关联起来的呢？它们之间的链接是什么呢？图 9-2 展示了我第一次尝试为迷宫游戏画的状态图。

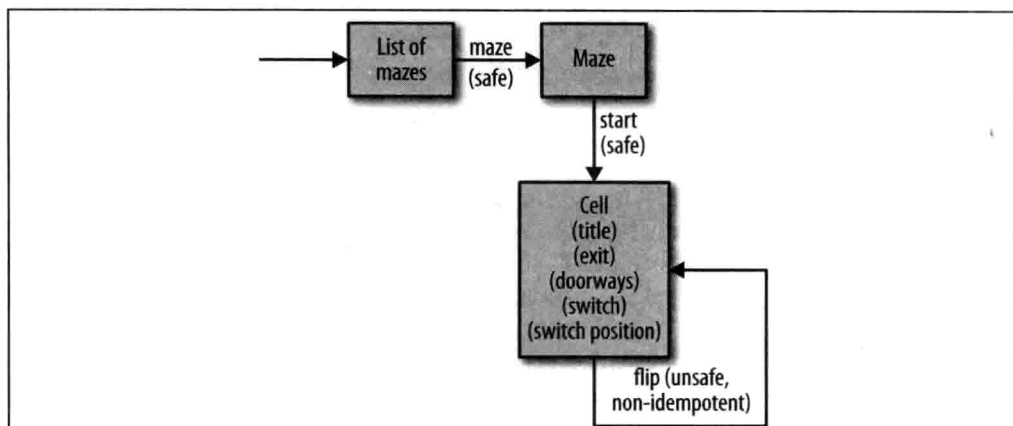


图9-2 迷宫游戏的状态图

有一些链接是显而易见的。在图 9-1 的层级视图中，如果一个方框完全包含另一个方框，那么这两个表述很可能是有关系的。无疑，迷宫列表和单个迷宫之间应该有一个链接，而迷宫与迷宫的入口单元格之间也应该有一个链接。现在，我将这两个链接的链接关系称为 `maze` 和 `start`。

语义描述符可能变为链接关系

一旦你获得了一张拥有方框和箭头的状态图，一切都变得很明显了：你的某些语义描述符实际上是安全的状态转换的名称。看看图 9-2 中的状态图，就很清楚了，“链接两个迷宫单元格的门口”不是一条独立的数据。它是一个链接：两个单元格之间的关联关系。类似的还有，“当前迷宫的出口”也不是一条数据。它是迷宫单元格与某个没有在该图中显示的事物之间的链接。这意味着，`north` 和 `exit` 都不应该是语义描述符：它们应该是链接关系。

图 9-3 展示了修订后的状态图，在该图中，出口 (`exit`) 和门口 (`doorway`) 都是作为链接而非数据进行展示的。

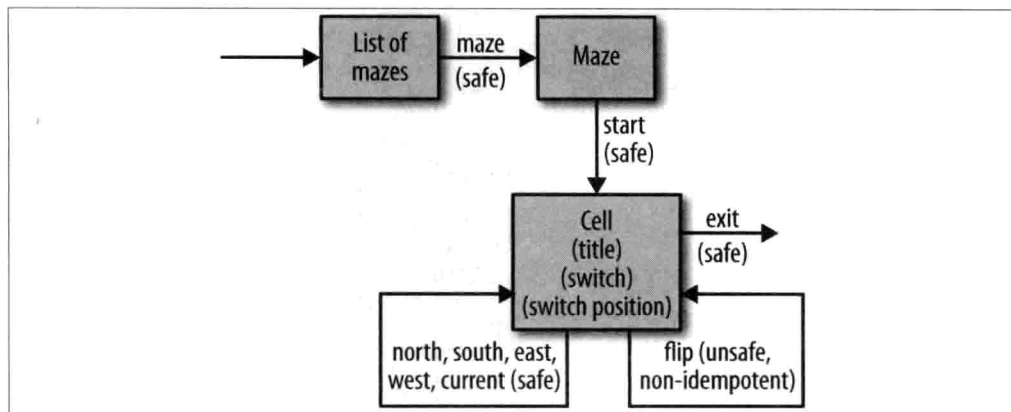


图9-3 修订后的迷宫游戏状态图

现在，这个迷宫的状态图就像一个真正的迷宫一样——同图 5-3 进行比较。这不是一个非常让人感兴趣的迷宫。穿越迷宫的所有乐趣——向北走，然后向东，然后继续向东——都被抽象成一个连接两个单元格表述的箭头。但是重要的是，客户端所能发起的每个 HTTP 请求都可以表示成沿着某个箭头从一个表述到达另一个表述的旅行。而这是你在图 9-2 中所不能言明的。

我下一步会将更多的语义描述符转换成链接关系。图 9-4 展示了一个稍微不同于图 9-1 的分割方式，它将表述分成资源：我在开关（switch）周围另外画了一个方框。在图 9-4 中，开关是它自己的资源，它独立于包含该开关的迷宫单元格。字符串 `switch` 之前被当作一个语义描述符，但是现在，它是一个链接关系，它指向作为独立资源的开关。图 9-5 展示了这些改动在状态图中的反映。

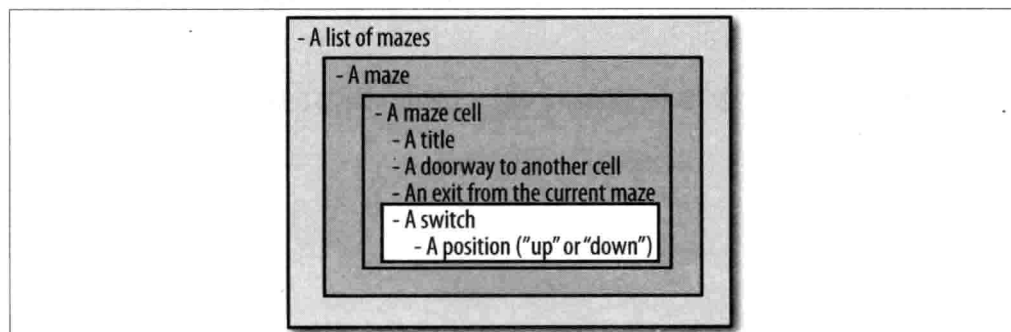


图9-4 将开关作为独立资源进行划分

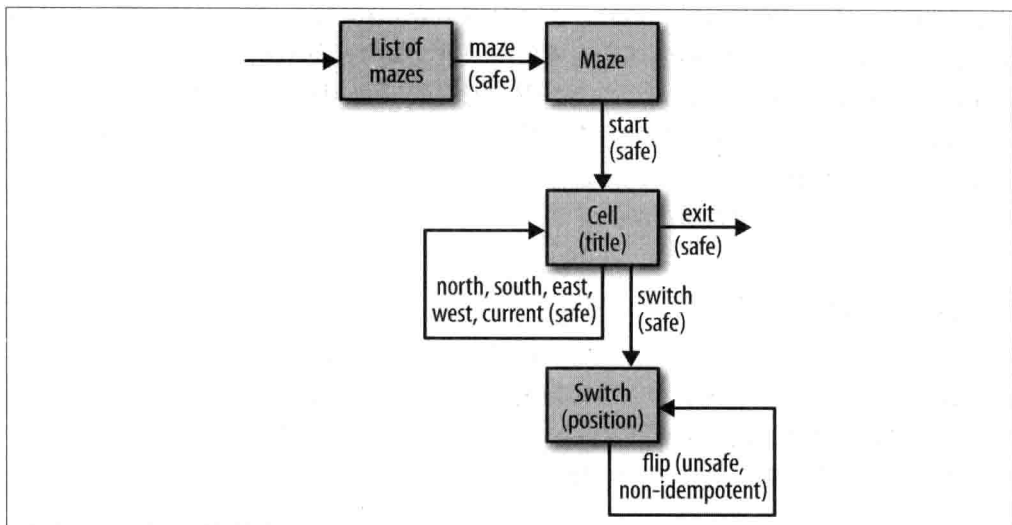


图9-5 开关成为独立资源后的状态图

我之所以做出如此改动是由于开关支持一种迷宫单元格所不支持的状态转换 (flip)。这就是说客户端和服务端应该能够将开关本身作为一个事物来进行对话通信。正如我在第 3 章中讲过的那样，任何能够成为客户端 - 服务器端的会话主题的重要事物都应该成为一个拥有自己的 URL 的资源。将开关作为一个独立的资源来处理会消除任何关于：HTTP 请求是在访问迷宫单元格还是开关，这一问题的歧义。

如果你希望的话，我可以更进一步。我可以将迷宫单元格的标题变成一个拥有自己的 URL（比如 `/cells/1/title`）的独立资源。在这种情况下，`title` 就从一个语义描述符变成了链接关系，正如 `switch` 所做的那样。

从技术角度而言，这没有错，但是我并同意这么做。开关是迷宫单元格的一部分，但是我们并不能将单元格作为一个整体来应用开关所支持的状态转换 (flip)。这也就是将开关作为一个独立的资源处理更加有意义的原因。而单元格的标题仅仅是单元格的部分信息而已。如果没有强有力的性能原因而使得必须将这个信息单独提供的话，我不认为标题应该成为一个独立的资源。

设定主页

你的状态图应该包含一个从外部某个位置进入的箭头。这个箭头代表的是客户端对你的告示牌 URL 发起的第一个 GET 请求。其他的每个箭头都必须源自于其中的某个表述，每个表述都应该可以通过状态转换从最开始的表述中访问到。

如果你第一步建立的层级结构中有一个明显的顶级对象的话，那么你就会拥有一个明显的候选对象来作为你的顶级表述。在本例中，这个最上层的表述就是迷宫列表。如果不存在一个明显的顶级对象的话，你就应该创建一个顶级对象。它并不需要精心设计。你仅仅需要一个安全的状态转换的列表就够了，这些状态转换可以链接其他重要的表述以及执行一些查询操作。你也可以添加一些不安全的状态转换进去，这些状态转换可以完整一些重要的工作，比如创建新的资源。

第3步：调整命名

从技术角度来说，你可以跳过这一步。不论你给那些充满魔力的字符串怎么命名，你的 API 都是拥有相同设计的。但是命名对人类而言还是比较重要的。虽然计算机是你的 API 的消费者，但是它们还是要代表人类来工作，人类需要理解那些充满魔力的字符串的含义。我们就是这样来消除语义鸿沟的。

许许多多的人已经花费了数以百计的人年（person-years）的工作量为各个领域的问题提出了各种 profile（请阅读“语义动物园”一章）。这些 profile 传达了必要的应用语义用于展示人、团体、公司、事件、产品、支付方式、地理位置、地标建筑、书籍、电视节目、工作列表、健康状况、日志、食谱，等等。这些 profile 并不包括人类在线上的交互活动（the online interactions），比如加入群组、离开群组、对事件的反应、为视频“点赞”，等等。那些最通用的、复用最频繁的 profile 都榜上有名，它们都已经升格到 IANA 注册链接关系列表中。

我建议你花一些时间来浏览这些 profile，来看看有哪些你能够复用的名称。尽可能地复用已有的名称可以降低人们误解你的魔力字符串的机会。它也会减少你需要编写的资料的数量，因为你可以复用那些定义了这些名称的 profile。它同时也增加了客户端开发人员能够复用现有的代码库的机会。此外，它也降低了你将来需要改变名称的可能性。

固然，大部分现有的 profile 都和某个具体的媒体类型相绑定，我也认为仅仅为了能够使用某个 profile 而选择一个媒体类型的想法是很糟糕的。这也就是我提出 ALPS 并在第 8 章中花了很大篇幅对其进行介绍的原因。ALPS 可以将这些 profile 从它们的媒体类型中解放出来。

Siren 文档不能使用 schema.org 的微格式 profile 来描述书籍。但是它可以使用根据 schema.org 的 profile 建立的 ALPS profile 来实现这样的目的。同提出一个全新的 Siren profile 相比，这可以减少很多的工作，而且这也增加了你的用户已经熟悉所提供的应用语义集合的机会。

现在将这一内容带到我的迷宫游戏的 API 中：我将我的方向链接关系称为 north、

south、east 和 west，这是因为它们是 Maze+XML 格式所采用的名称。即便我在第 4 步中不再选择 Maze+XML 作为我的媒体类型，认识到下面这一点也是很有用的，那就是：人们已经考虑过这个问题并认为 north 是一个比 n 更好的名称。此外，多亏了 alps.io 上的 ALPS profile 描述了 Maze+XML 的应用语义，我可以复用一些 Maze+XML 的应用语义而不需要采用 Maze+XML 媒体类型。

几乎所有面向消费者的 API 都可以通过这种方式复用一些语义。最明显的例子是，你不需要提出自己的词汇表来描述人类的个人信息。这个领域已经被 hCard、schema.org 的 Person 以及 FOAF 都覆盖到了。

这种覆盖在一些专业领域，比如经济、法律、甚至于软件开发等做得并不够好。那些词汇更倾向于从普通消费者而非实际工作者的视角来进行定义。相较于语义描述符，链接关系的覆盖在这方面就更差了。

比如，在 schema.org 中有一项叫作 <http://schema.org/Offer>。它描述的是销售物品的意向，它定义了诸如 price（价格）、warranty（保修单）以及 deliveryLeadTime（交货时间）等语义描述符。但是它没有定义让客户真正购买物品的不安全的链接关系。在这种情况下，你可能使用 purchase 链接关系，这个链接关系是由 alps.io 定义并从 Activity Streams 标准中摘抄的。或者，你也可以自己创建一个你自己的名称用于该链接关系。

如果你认为我的在 API 之间复用语义的观点非常可笑和不切实际，或者为了复用语义描述符，有太多的清除和打扫工作要做，那么你也可以自由地为每个事物构造自己的名称。正如我前面所讲的，从技术层面讲，名称一点也不重要。但是我有两条建议的底线你应该保证遵循。

第一，不要根据你的数据库模式或者对象模型的字段来自动生成你的语义描述符的名称。否则，这会使得你的客户端从软件层次上依赖于服务器端的代码。当你改变了那些代码了以后，除非你通过你的 API 来引入一个兼容层用于展示那些旧名称，否则你的客户端会一直不能正常运行。

第二，不要提出和 IANA 注册的链接关系在功能上重复的链接关系。那些 IANA 注册链接关系被特别记录到注册表中就是由于它们并不和某种媒体类型或者应用领域关联。它们是最通用的应用语义，为了便于复用，它们全都被列到一起了。

如下是一些具体的例子：

- 任何时候，如果你得到的是事物的列表与列表中的单个事物之间的关联关系，你就应该考虑使用 IANA 注册链接关系 collection 和 item，而不要使用（或者补充）一些其他更具体的内容。

- 有两种方式可以用于对跨多个表述的资源状态进行分页。其中明显的方式也就是在网站中很常见的方式：采用链接关系 `first`、`last`、`next` 和 `previous`（或者 `prev`）。另一种就是 RFC 5005 所描述的以档案归档为基础的技术，它使用的链接关系是 `current`、`next-archive` 和 `prev-archive`。除非你提出第三种分页技术，否则你没有理由为这些关联关系构造新的名称。
- 你可以使用 `replies` 关系来描述留言主题（message threads），这个关系最早是由 RFC 4684 为 Atom 定义的。
- 如果你希望保留资源的历史状态，你可以使用 RFC 5829 定义的 `latest-version`、`successor-version`、`predecessor-version`、`working-copy` 和 `working-copy-of` 来将不同版本的状态链接起来。
- 链接关系 `edit` 和 `edit-media` 都是足够通用的可以适用于许多不安全的状态转换。如果你得到的状态转换除了更新一些资源状态外什么也不做，你也许能够将它称作 `edit` 而不要使用某些更加具体的词语。

下面我为你展示一个用 `edit` 来进行替换的例子。在迷宫游戏中，链接关系 `flip` 会改变开关的位置。如果开关是 `up`，`flip` 转换就将它设置为 `down` 状态，如果开关是 `down`，也是同样的。它不是一个安全的转换，并且，它也不是幂等的。触动两次开关和触动一次开关的结果是不相同的。

如果我们不采用 `flip` 而将链接关系称为 `edit`，会怎么样呢？不同于之前相对于开关的当前位置来改变位置，现在由客户端来决定它所想要的位置——`up` 或者 `down`——并在触发 `edit` 转换的时候将这个信息发送出去。API 的状态图就如图 9-6 所示。

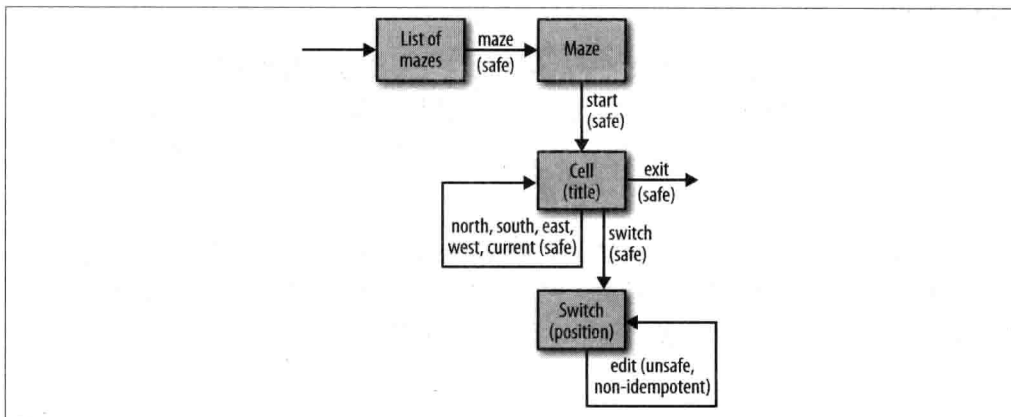


图9-6 采用IANA的“edit”来代替“flip”后的状态图

用 `edit` 来代替 `flip` 有两个好处。客户端不再需要学习一个全新的链接关系，而可以复

用它已有的关于 IANA 注册链接关系 (edit) 的知识。它只需要学习的是一个自定义的语义描述符 (position)，而如果客户端想要读取一个表述的话，这个描述符是不管怎样都必须学习的。

第二，状态转换现在是幂等的。两次将开关设置为 down 的效果和只一次将开关设置为 down 的效果是相同的。如果某个超媒体文档描述了这个使用 PUT 请求的 edit 转换，那么客户端就知道它可以在第一次请求执行失败的情况下，重试这个请求。

使用 edit 关系的 HTML 超媒体表格内容如下：

```
<form action="/switches/4" method="POST">
  <input type="radio" name="position" value="up" default="default"/>
  <input type="radio" name="position" value="down"/>
  <input type="submit" class="edit" value="Set the switch!"/>
</form>
```

(这里的 method 是 POST，不是 PUT。这是由于 HTML 表单不能使用 PUT。所以一个采用如上 HTML 表单的客户端不能利用 edit 是幂等的这一事实上的优势。但是相对于 flip，客户端仍然更有可能理解 edit。在 Siren API 中，edit 状态转换可以用 PUT 进行表示，而客户端就可以知道这个状态转换是幂等的。)

如果你想要 flip 那种老式的、非幂等的行为方式，你可以用 edit 来进行模拟。你可以通过提供一个不同的超媒体表单来这么做，这个表单会触发相同的状态转换，但是它不允许客户端来改变预定的 position 值：

```
<form action="/switches/4" method="POST">
  <input type="hidden" name="position" value="off"/>
  <input type="submit" class="edit" value="Flip the switch!"/>
</form>
```

客户端在激活这个控件后，会收到一个新的表述，这个表述包含了一个不同的控件：

```
<form action="/switches/4" method="POST">
  <input type="hidden" name="position" value="on"/>
  <input type="submit" class="edit" value="Flip the switch!"/>
</form>
```

在这个设计中，edit 链接关系总是幂等的，但是客户端激活它所看到的每个 edit 控件时会触发非幂等的状态转换。

就我而言，所有的这些设计都是 RESTful 的。它们都使用了超媒体来描述状态转换。大家更青睐于 edit 多过 flip 的唯一的原因是大家都已经在 edit 的含义上达成了共识。

第4步：选择一种媒体类型

既然你已经得到一些能够满足你的业务需求的语义，现在是时候来选择一种超媒体格式来表示它们了。它很可能是我在第 6、7、10 章中提到的那些超媒体类型中的一种。你也可以自由地设计一种新的领域特定的媒体类型，尽管你不需要这么做。

168 虽然没有一种媒体类型能够永远成为最佳选项，但是在目前，还是有一些常见的模式出现。如果你的状态图类似于图 9-7，那么你的协议语义就是实现了集合模式。你应该考虑采用 Collection+JSON、AtomPub 或者 OData（见第 10 章）。

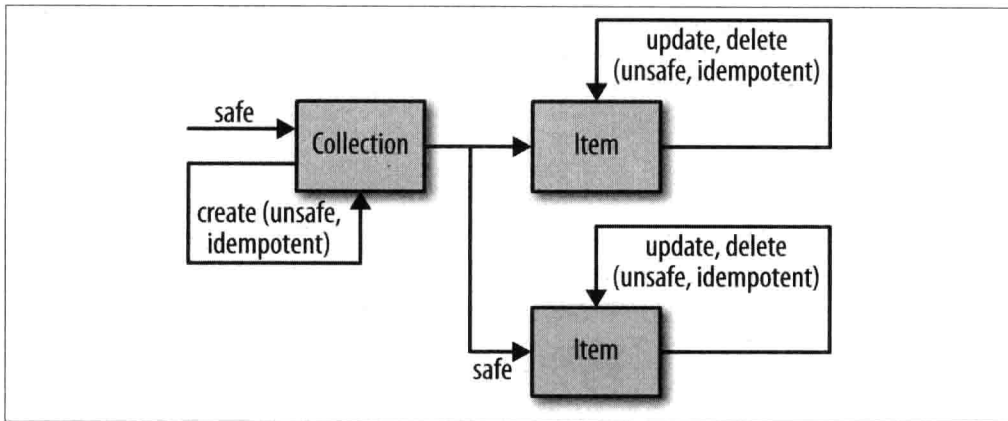


图9-7 集合模式的通用状态图

如果你的状态图看起来更像图 9-8 那样比较杂乱，你很可能会希望采用一种通用的超媒体语言：HTML、HAL 或者 Siren。

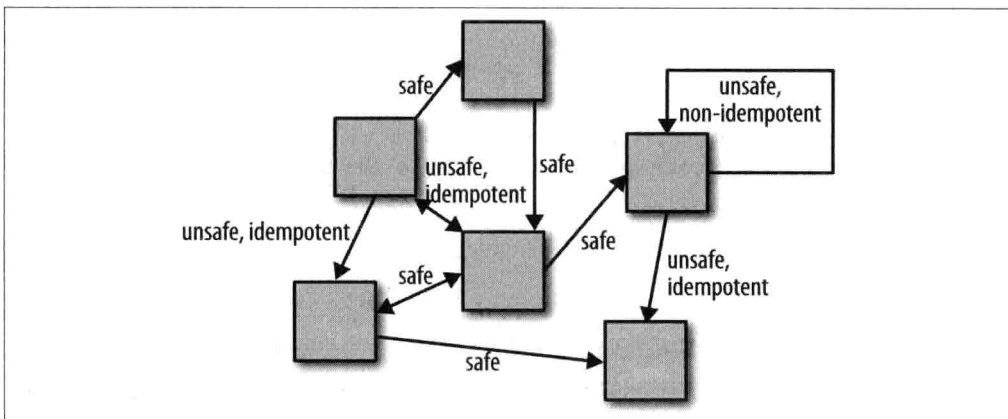


图9-8 超媒体的任务

如果在这种情况下,你正在考虑使用 JSON,我必须提醒你,JSON 并不是一种超媒体格式。JSON 标准定义了诸如数字、列表、字符串以及对象等概念。它并没有定义链接或者链接关系的概念,所以,它不具备超媒体的能力。你需要的是一些更加具体的格式:可能是 HAL、Siren、Collection+JSON 或者 Hydra。

如果在这种情况下,你正在考虑使用 XML,当然,XML 是一种超媒体格式,这是由于你可以使用 XLink 和 XForms (见第 10 章)来为任何的 XML 文档提供类似于 HTML 的功能。但是我觉得那不是你真正想要的。你可能需要的是更加具体有针对性的格式:HTML、HAL、Siren 或者 AtomPub。

如果你想要同时提供 JSON 和 XML 格式的表述来表示相同的语义,你应该选择一种能同时提供这两种格式的超媒体标准。这就意味着应该选择 HAL 或者 OData,尽管 Siren 的 XML 版本也已经在规划中了。

如果你的 API 是只读的——也就是,你的状态图不包含任何不安全的转换——你会有很多好的选择。我的建议有:HTML、HAL 或者 JSON-LD。

如果你的 API 确实包含一些不安全的转换,这就会限制你的选择范围。JSON-LD 格式不能仅仅靠自己表示不安全的状态转换;你需要增加 Hydra (见第 12 章)。HAL 支持不安全的转换,但是我认为它的效果并不好。

Collection+JSON 仅仅支持特定的不安全的转换:添加一个新的子项到集合中、编辑某个子项、删除某个子项。这就是它全部的内容。你不能使用该媒体类型定义的这三种不安全的转换之外的任何不安全的状态转换。

第5步:编写Profile

当你的服务器发送了某个表述时,这个表述会包含一个叫作 **Content-Type** 的报头,这个 header 会告诉客户端来如何解析这个表述。你也可以包含一个指向一个或者多个 profile 的链接,这些 profile 会说明这个表述的应用语义。

在第 3 步中,你很可能找到了一些已有的能覆盖你的部分应用语义的 profile,但是很可能它们不能将所有的应用语义都覆盖到,还会有一些关于你的 API 的特殊的应用语义没有被涉及到。你需要编写一个新的 profile 的方式来说明这部分特殊的语义。

如下的标记代码片段表示第 7 章的迷宫游戏导入了两个 ALPS profile。其中一个来自于 alps.io,它定义了 Maze+XML 的语义;另一个自定义的 profile 说明了我为 Maze+XML 所唯一增加的那一部分内容:神秘的开关。

```
<link rel="profile" href="http://alps.io/example/maze"/>
```

```
<link rel="profile" href="/switches.alps"/>
```

你的 profile 可以是一个 ALPS 文档、一个 JSON-LD 上下文或者是采用了 XMDP 微格式的网页。如果所有这些选项都不适合你，你可以放弃机器可读的 profile 的想法，取而代之的是编写一个人类可读的 profile。

人类可读的 profile 看起来类似于传统的 API 资料文档。它是一个清晰地展示了所有的链接关系和语义描述符的含义的网页。你还是可以复用那些来自于 alps.io 以及其他 profile 的链接关系和描述符的，只不过是采用复制、粘贴的方式将那些文本添加到现有的 profile 中，并确认增加链接回原始 profile 的链接。

第6步：实现

你会在这一步花费你大部分的时间，我也不会提供太多建议，因为这一步依赖于你所采用的框架和编程语言。如果你拥有一张状态图以及一份感到满意的 profile 的话，那么这一步的工作恐怕不会轻而易举地完成，但是它至少应该是有明确方向的。

第7步：发布

如果你足够幸运，你找到了已有的和你的需求完全匹配的领域特定的标准，而且不需要任何的扩展，在你做完第 6 步的工作后，你的工作就算完成了。你的 API 资料包含一个 URL（你的 API 的“告示牌 URL”）以及 Content-Type 报头的一个值就足够了。

但是这几乎不可能发生。为了满足你的业务需求，你不可避免地需要对已有标准进行扩展，或者设计一套全新的标准。到了这一步，你已经设计了那“新增”的部分，并采用机器可读的结构（比如 ALPS profile）以及人类可读的文档（比如媒体类型的定义）的组合方式对它进行了说明。剩下的仅有的工作就是将这些信息进行发布。

这要比仅仅将 API 资料放到你的网站上要复杂多了，但是它又不是那么复杂。我下面就将这一步进行展开介绍以避免你不小心跳过某一步。

发布你的告示牌URL

回到第 2 步，我曾经说过你的状态图应该拥有一个方框来让一个来自于外部某个地方的箭头指过去。这个方框代表的就是你的主页：通往你的所有其他资源的超媒体网关。每个想要为你的 API 编写客户端的人都必须知道你的主页的 URL。剩下的内容都是可以协商来获取的。

如果你没有设计一个“主页”资源就结束了第 2 步，我建议你回到第 2 步，设计一个“主页”资源。你的告示牌 URL 是关于你的 API 的信息中最重要的一个，因为它是通向其

他所有资源的门户。

发布你的profile

你的 profile 文档是和其他关于你的 API 的信息一起放在你的网站上的。如果你已经编写了一份 ALPS profile，并将该 profile 注册到 alps.io 上的 ALPS Registry 中，那么我会对你这样的行为表示赞赏。这会有助于其他人找到和复用你定义的链接关系和语义描述符。

注册新的媒体类型

你很可能并不需要设计一种新的媒体类型，所以我不会占用多少篇幅来说明当你完成了这样的设计工作以后应该怎么办（我会在后面一节“设计媒体类型”中进行讲解）。一言以蔽之，一旦你完成了开发工作，你应该对自己的设计足够自信来将你新的媒体类型注册到 IANA 中。

注册新的链接关系

171

如果你的链接关系都是 URL (RFC5988 将其称为“扩展关系类型”)，你不需要做任何特别的工作。没有人能够定义一种和你的链接关系相冲突的关系，因为你是根据你所管理的域名来对你的关系命名的。如果你并不认为会有任何其他 API 提供者想要复用你的链接关系，你不妨省点事来使用扩展关系。

但是纵观全书，我在避免使用扩展链接关系。它们都太长了以至于不能在印刷的时候一遍遍地反复使用。和扩展链接关系不同，我使用的链接关系都是很短的字符串，比如 west 和 flip。RFC5988 将这些链接关系称为“注册关系类型”，并为了避免使用冲突，它们需要被注册到某个地方。如果你见到一份使用 rel="current" 的 HTML 文档，那么这个 current 代表的是一个集合中的当前的子项，还是电流的度量单位呢？这一点必须是明确无歧义的。

RFC5988 并没有确切地说明如何注册一个链接关系，但是我认为存在 4 种注册方式：

- 它可能会在 IANA 的链接关系注册表中找到。任何 API 提供方都可以在未定义的情况下在表述中使用 IANA 的注册链接关系。一个很好的例子就是 RFC4685 所定义的 replies 关系。
RFC 5988 的 6.2 节描述了 IANA 的注册流程。将一个链接关系添加到注册表中需要编写一份 RFC（或者等价的文档），并且只有那些通用的关系才会被接受，所以采取这条路线的开发人员相当少。
- 链接关系可以和媒体类型定义在一起，就像 Maze+XML 定义 west 和 exit 那样。一些其他的媒体类型定义的 exit 关系的含义可能并不同，但是谁在乎呢？一个文

档只能使用一种媒体类型，所以要应用哪些规则还是一清二楚的。

如果一种媒体类型定义的链接关系和 IANA 上注册的关系相冲突，媒体类型的定义优先。不要刻意这么做！我之所以强调这一规则是为了保证你的 API 的应用语义不会由于你的媒体类型使用的链接关系已经被其他人在 IANA 注册过而发生变换。

- 链接关系也可以定义在机器可读的 profile 中，比如 ALPS 文档。一些其他的 profile 可能以不同的方式来进行定义，但是谁在乎呢？这个文档并不使用那些 profile。

如果某个 profile 定义的链接关系和媒体类型中定义的关系或者 IANA 注册的关系相冲突，profile 的定义优先。再次强调，不要刻意地这样使用。这是一个“以防万一”的规则。

- 172
- 链接关系可以注册到 Microformat wiki 中^{注1}。Wiki 的网页不是排他的；它试图将所有发现或者提交的链接关系都编订目录用于 HTML。

Microformat wiki 是那些可能将来某一天会进入 IANA 注册表中的链接关系的很好的实验平台。如果你想要其他人来使用你发明的链接关系，将它放到 wiki 上是一个很好的测试方法。如果你不打算这么做的话，我建议你使用扩展链接关系作为代替。

你可以使用 ALPS 来作为折中方案。即便你没有将某个 ALPS 文档作为一个 profile 包含进去，你仍可以将该 ALPS 文档定义的所有的链接关系的完整的 URL 作为扩展链接关系来使用（比如 <http://alps.io/example/maze#exit>）。当你将该 ALPS 文档作为 profile 包含进去以后，你可以将它的链接关系当作注册链接关系（如 exit）来使用。

发布剩余的其他资料

还有许多的资料需要发布，但是这些全都是针对你的 API 的人类可读的文档：摘要、范例、代码示例、授权配置说明书、介绍你的 API 不同之处的市场推广资料。

这些资料都非常重要，但是你发布这些资料并不需要得到我的鼓励。这是我们在考虑 API 资料的时候所考虑的东西。我在这个过程中对人类可读的资料略过不谈，这是由于，从我的经验来看，它们经常是作为超媒体控件的一个替补而使用的。

软件客户端在适应超媒体文档的变化方面有一定的适应能力。基于人类可读文档而编写的软件则并不具备适应能力。如果一个 API 仅仅是用那些单调的白话文来描述的，那么这些文字的改变就意味着要重写所有的客户端。这是当前的 API 所面对的一个大问题，这也是我试图通过本书所缓解的问题。

注1 网页链接是：<http://microformats.org/wiki/existing-rel-values>。

我希望尽可能地加强客户端的适应能力。这意味着设计过程要严格关注于创建机器可读的文档，而人类可读的文档仅作为给用户提供的便利而出现。

Well-Known URI

想一想：如果你不需要推广你的告示牌 URL，会怎么样呢？如果客户端正好知道如何找到你的 API 的入口点会怎么样呢？这就是 IANA 的“*Well-Known URIs*”的约定，它是在 2010 年由 RFC5785 制定的。

如果服务器端采用 CoRE Link Format（见第 13 章）来展示表述，这就不需要来知道告示牌 URL 是什么了。这个 URL 总是 `/.well-known/core`。这个（相对）URL 是在 IANA 中注册过的。CoRE 客户端就不再需要针对每台服务器去了解其不同的告示牌 URL 了。它总是可以向 `/.well-known/core` 发送一条 GET 请求，然后获得一系列的超媒体链接，这些链接都分别链接到服务器端所管理的其他资源。提供 web host metadata 文档（见第 12 章）的服务器总是应该通过 `/.well-known/host-meta` 或者 `/well-known/host-meta.json` 提供该文档。

173

这是非常小的事情，但是它消除了最后的那点语义鸿沟。多亏了 Well-Known URI Registry，这使得在理论上成为可能：只提供主机名的情况下，客户端就可以研究和新的 API。

需要注意的是，这些 well-known URI 通常都是和特定的媒体类型关联的。正如我所写的，如果你没有使用 CoRE Link Format 或者 web host metadata，你就不能将你的 API 发布到某个 well-known URI。它们是 Well-Known URI Registry 上仅有的两种对 API 有用的格式。

实例：You Type It, We Post It

本节将花费一些时间来使用第 7 章中的迷宫游戏执行一遍我的设计流程，因为，我前面一节一直都只是在解释流程中的每一个步骤。现在换一种方式，我将采用一个简短的实例。我将仅仅展示出我的决策而不再解释所有的步骤。我的问题域是第 1 章中提到过的“*You Type It, We Post It*”网站。我将只完成前 5 个步骤，以一个设计和一份 profile 来结束，而不再进行编码实现。

罗列语义描述符

通过查看第 1 章中关于该网站的描述，我识别出了下列的语义描述符：

- The home page
 - Some kind of “about this site” text

- The list of messages
 - An individual message
 - The ID of a message
 - The text of a message
 - The publication date of a message

然后，我将这些描述符以一种我认为合理的方式进行分组，结果如图 9-9 所示。我获得了 3 种不同的表述：“about this site”的文本、消息列表以及一条单独的消息。我决定将消息列表作为“主页”来使用，而不再使用那个只有两个链接分别指向消息列表和“about this site”文本的单独的主页。

174

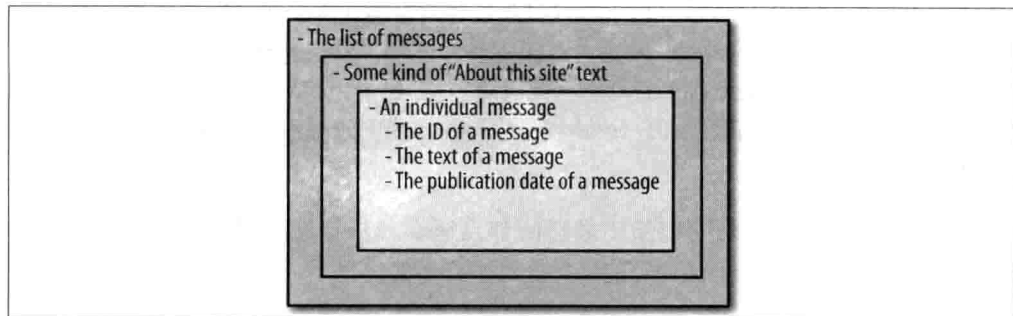


图9-9 对You Type It...的语义描述符进行分组成为表述

画状态图

图 9-10 展示了我所提出的状态图。

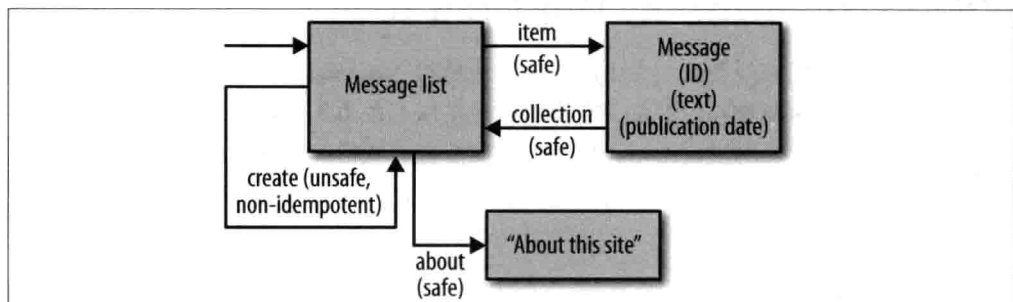


图9-10 You Type It...API最初的状态图

以 You Type It...网站上的链接为导向，我使用安全的状态转换将 3 种资源连接了起来。我同时创建了一个不安全的状态转换，对应于网站上创建一条新的消息的 HTTP POST 表单。

调整名称

当我为状态图的安全的状态转换进行命名时，我确信要选择 IANA 注册的名称：`about`、`collection` 和 `item`。而“`about this site`”的文本是一个人类可读的文档，所以我不担心它的语义描述符。

还有 6 个 IANA 无能为力的事物需要命名：“`list of messages`”、“`message`”、“`create`”、“`ID`”、“`text`”和“`publication date`”。请浏览一下“语义动物园”一章，这会对其中的 5 个事物的命名有所帮助。

Schema.org 中有一个微数据项被称为 `BlogPosting` (<http://schema.org/BlogPosting>)，它定义了称为 `articleBody` 和 `dateCreated` 的语义描述符。这能分别对应到“`message`”、“`text`”和“`publication date`”。Schema.org 的 `BlogPosting` 的集合被称为 `Blog`。这个也就照应了“`message list`”。

◀ 175

我将我的那个不安全的状态转换命名为 `post`。我是从 Activity Streams 标准中借鉴了该名称，在该标准中，`post` 的含义是“创作某个对象然后将其发布到网络的行为”。还没有人曾经将 schema.org 的微数据和 Activity Streams 的动词有意地放到一起工作，但是 ALPS 使得我可以将它们的应用语义结合起来。

现在就只剩下 `message ID` 了。我认为我并不真的需要提供这个信息。每个消息已经拥有一个唯一的 ID：它的 URL。客户端为什么应该要关心服务器所使用的内部 ID 呢？所以我决定将它从我的 API 中删除。

图 9-11 展示了名称调整之后的状态图。请注意：`item` 链接现在拥有两个链接关系：`item` 和 `blogPost`。第二个链接关系来自于 schema.org 的 `Blog` 条目，它将 `Blog` 和 `BlogPost` 之间的关联关系定义为 `blogPost`。这和 IANA 中更加通用的 `item` 关系有点重复，但是没有理由说我不能在一个链接上附加两个链接关系。这样，理解 schema.org 的 `Blog` 和 `BlogPost` 的客户端就不再需要同时理解 IANA 的 `item` 了。

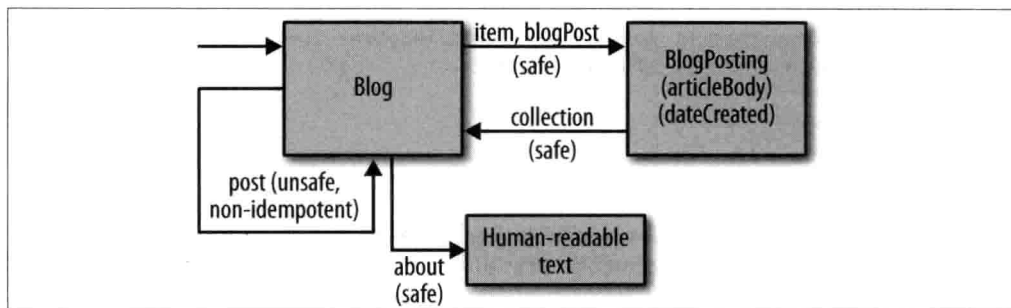


图9-11 名称调整后的You Type It...API的状态图

我正在创建第 58 个微博 API 吗？从某种意义上说，是的。但是我没有定义任何新的东西。我所采用的所有东西都是来自于 IANA、schema.org 以及 Activity Streams。一个已经理解了这些语义描述符和链接关系的客户端就会理解我的 API。这样的客户端不大可能存在，但是在我为了这第 58 个 API 而重新设计了这些基本概念以后，和以前相比，这样的客户端的组成部分已经存在的可能性就大大增加了。

选择一种媒体类型

我可以在数量庞大的媒体类型中进行选择。我的状态图类似于图 9-7，所以一种实现了集合模式的媒体类型会有很大帮助。第 2 章中所展示的那个真实的 YouTypeItWePostIt.com API 采用的是 Collection+JSON。我也可以走纯超媒体的路线：YouTypeItWePostIt.com 网站所采用的 HTML。我甚至可以选择一种领域特定标准。在第 6 章中，我将 AtomPub 作为一种通用的“集合模式”来对待，但是它最初定义的目的是专门用于发布独立的文本。

我的选择可能改变我所使用的词汇表。如果我选择 Atom 作为我的表述格式，我就需要停止将消息的文本部分称为 `articleBody`，而开始将其称为 `content`，因为这才是 Atom 对它的叫法。

仅仅为了打破常规，我将要采用 HAL。消息列表的 HAL+XML 表述可能如下所示：

```
<resource href="/">
  <link rel="profile" href="http://alps.io/schema.org/Blog"/>
  <link rel="profile" href="http://alps.io/schema.org/BlogPost"/>
  <link rel="profile" href="http://alps.io/activitystrea.ms/verbs"/>
  <link rel="about" href="/about-this-site">

  <Blog>
    <link rel="post" href="/messages"/>

    <resource href="/messages/2" rel="item">
      <BlogPost>
        <articleBody>This is message #2.</articleBody>
        <dateCreated>2013-04-24</dateCreated>
      </BlogPost>
    </resource>

    <resource href="/messages/1" rel="item">
      <BlogPost>
        <articleBody>This is message #1.</articleBody>
        <dateCreated>2013-04-22</dateCreated>
      </BlogPost>
    </resource>
  </Blog>
```

```
</resource>
```

这个表述传递了所有必要的资源状态（消息列表中的两条消息的描述说明）并且包含了所有必要的超媒体链接（使用链接关系 `profile`、`about`、`item` 和 `post`）。

`post` 链接还是有一个问题：我们不清楚这是不是一个应该由 `POST` 来触发的不安全的状态转换，我们也不清楚客户端应该在发送 `POST` 请求的时候发送什么实体消息体。但是这要归结于 `HAL` 的一个具有普遍性的缺点。如果我不喜欢 `HAL` 的这个特性，我可以在这时选择另外一种不同的媒体类型。

编写Profile

由于我所有的应用语义都是从现有的 `profile` 中借鉴的，所以从技术角度来说，我并不需要编写我自己的 `profile`。下面的表述示例仅仅是链接到 3 个已有的 `ALPS profile`：

```
<link rel="profile" href="http://alps.io/schema.org/Blog"/>
<link rel="profile" href="http://alps.io/schema.org/BlogPost"/>
<link rel="profile" href="http://alps.io/activitystrea.ms/verbs"/>
```

◀ 177

它囊括了除 `IANA` 的链接关系 `about` 和 `item` 之外的所有内容。对于这两个链接关系，我可以不加说明直接使用它们。

下面是为我的 “You Type It, We Post It” 而设计的一个独立的 `ALPS profile`（它的有些内容是多余的，但是它是一个包含了我的 `API` 真正使用的所有链接关系和语义描述符的文档）：

```
<alps>
  <descriptor id="about" type="semantic"
    href="http://alps.io/iana/relations#about"/>

  <descriptor id="Blog" type="semantic" href="http://alps.io/schema.org/Blog">
    <descriptor id="blogPost" type="semantic"
      href="http://alps.io/schema.org/Blog#blogPost" rt="#BlogPosting"/>
    <descriptor id="item" type="semantic"
      href="http://alps.io/iana/relations#item" rt="#BlogPosting"/>
    <descriptor id="post" type="unsafe"
      href="http://alps.io/activitystrea.ms/verbs#post">
    <descriptor href="#BlogPosting"/>
  </descriptor>
</descriptor>

<descriptor id="BlogPosting" type="semantic"
  href="http://alps.io/schema.org/BlogPosting">
  <descriptor id="articleBody" type="semantic"
    href="http://alps.io/schema.org/BlogPosting#articleBody">
  <descriptor id="dateCreated" type="semantic"
```

```
href="http://alps.io/schema.org/BlogPosting#dateCreated">
</descriptor>
</alps>
```

不再需要提供 3 个 profile 链接，现在我的表述只需要链接到这个 profile 就够了。

不懂我的 profile 的客户端可以将我的表述作为纯粹的 HAL 表述来处理。这没有什么用处，因为就其本身而言 HAL 没有定义任何协议或者应用语义。HAL 浏览器可以解析我的表述，并且从数据中识别出链接，但是它无法知道这些链接以及数据的含义。

设计建议

希望到此为止，你已经清楚地了解了我是如何执行这个设计流程的。现在我希望能提出一些我在开发和应用这一流程中所总结的具有一定实用性的经验教训。

178 资源是实现的内部细节

大部分设计 RESTful web API 的流程关注的都是资源设计。但是在这里并没有资源。状态图中的方框并不是资源，它们是资源的表述——那些在客户端和服务端来回传送的实际文档。

这不是疏忽，资源是 HTTP 的基本内容，并且对于 API 实现来说非常重要，但是我逐渐认识到，它们自己对 REST 而言并不是特别重要。我的设计流程关注的是状态转换和语义描述符。一旦你确定了这些内容，你也就拥有了资源。

回想一下 HTTP 在客户端和服务端之间来回传送的情形。一个资源收到一个 GET 请求，然后会提供某种媒体类型的表述。这个表述包含了用于描述可能的状态转换的超媒体控件。客户端通过向另一个资源发送 HTTP 请求来激活这个状态转换，这个资源会实现该状态转换并发送另外一个表述回去。客户端从来不会直接与某个资源进行交互。

如果你到了第 6 步发现你需要实现一些之前没有预料到的资源，恐怕是你之前跳过了某一步。你所想到的这个资源需要从一些现有的资源链接过去。而这个链接就是一个状态转换，它应该出现在你的状态图中某个箭头上（第 2 步）。如果资源自己管理自己的状态，那么它需要一个表述。这个表述应该在你的状态图中作为一个方框显示出来（第 2 步）。这个表述必须拥有一种媒体类型（第 4 步），并且可能还需要一个 profile（第 5 步）。它传送给客户端的数据也应该显示在我在第 1 步中建立的那个很大的语义描述符列表中。如果这个资源支持任何一种不安全的状态转换，这些转换就应该已经在第 2 步中显示出来了。它们必须由内嵌在某些表述中的超媒体控件来进行描述。

如果你还没有确定一个资源的协议语义和应用语义的话，你就还没有真正设计这个资源。

如果你已经确定了这些内容，也就没有其他内容需要设计了。

以资源为第一重点不会给你带来坏的设计，但是它却倾向于从服务器端实现的角度来表达一个设计，而不是从客户端的体验角度。我们也很容易用良好的资源设计（由大量人类可读的资料作为支持）来作为避免考虑超媒体的一个借口。

不要掉入集合陷阱

虽然集合模式非常通用并且功能强大，但是它包含一个陷阱。我在过去这些年里见过太多人一次次掉进了这个陷阱中。我仔细思考后的建议就是不要上钩。不要使用你的数据库模式来作为你的 API 设计的基础。你要画一个状态图来代替。

乍一看，使用数据库模式好像是一个非常棒的主意。一个 SQL 数据库，拥有的 4 种基本的命令（SELECT/INSERT/UPDATE/DELETE）很自然地映射到 CRUD 模式（create/retrieve/update/delete），而 CRUD 模式也很自然地映射到 API 的集合模式，这也就很自然地映射到 4 种主要的 HTTP 方法（GET/POST/PUT/DELETE）。并不存在什么技术性的问题使得你不能跳过大部分流程步骤，而直接通过集合模式发布你的数据库模式。那么这会导致什么错误发生呢？

179

由于现代工具的帮助，这种策略可以让你很快得到一个可以工作的 API，但是它有两大问题。第一个问题源于一个事实，那就是你的用户并不关心你的数据库模式，他们关心的是你的应用语义，这两个仅仅是略有关联。你不会搭建一个仅仅是和数据库交互的原始界面。你应该就像设计网站一样将同样的思想应用于 API 设计中。

换句话说，你确实关心你的数据库模式——你保留根据你的需求变更而改动数据库模式的权利。这就是第二个问题。当你发布了基于数据库模式的 API 以后，改变数据库模式就基本上变成不可能的事情了。你已经给许多你从未听说过的人们添加了一层对你的数据库模式的软件依赖。那些人就是你的客户，你有责任来支持他们。数据库模式的改动、甚至那些网站用户都不会注意到的改动都会给你的 API 用户带来很大的问题。

有许许多多的技术用于处理这些问题，我将会在后面的一节“当你的 API 改变时”讨论这些内容。但是最好的策略就是第一时间避免这种情况的发生。

这也是我的设计流程中采用状态图的原因。远离这个陷阱是我所关注的一个很大的问题。考虑状态转换会迫使你认真考虑你的应用，而不是包含所有资源状态的数据库。

我并不是说避免使用集合模式。它是一个很棒的模式。如果你的状态图看起来类似于图 9-7，大胆地去使用它。但是首先要画状态图。不要将你的应用所定义的协议语义与你的数据库所定义的接口相混淆。

不要从表述格式着手

你可能忍不住想要在开始第 1 步之前就先选择一种表述格式，这样你就可以在理解了语义的时候设想出你的文档的样式了。使用一种通用的格式比如 HTML 来随手书写是不错的，但是我建议你推迟做出决定，直到你已经到达了第 4 步。这是因为表述格式不仅仅是被动的数据容器。它们还将关于协议语义和应用语义的前提假设引入到了每个使用它们的 API 中。这些前提假设可能和你的业务需求相冲突。

180 ▶ 举一个非常简单的例子，假设你选择使用 Maze+XML 来开始你的 API 设计，而原因仅仅是它是我第一个详细讨论过的表述格式而已。你很可能在犯错误。Maze+XML 定义了一个非常特殊的类似于图 9-6 的状态图。它定义了一系列的应用语义，在这些语义中，GET 请求表示“进入迷宫”、“从某个方向移动”、“离开迷宫”。简而言之，Maze+XML 是为迷宫游戏而设计的。

直到你完成了前两个步骤以后，你才可以选择 Maze+XML 格式。你需要将你的业务需求分解为一系列的协议语义和应用语义。如果能够证明它们符合 Maze+XML 定义的语义，那就太棒了。但是它们很可能不符合。

再举个例子，假设你决定为你的 API 使用 Collection+JSON（或 AtomPub 或 OData）进而开始设计 API。这就意味着你选择了一系列贯彻集合模式的语义。你这是在预先声明你的 API 的状态图和图 9-7 类似。

你如何才能提前声明这些内容呢？如果能够证明你的协议语义符合集合模式（通常都会符合），那么集合模式标准就是你想要的了。但是你不能凭直觉来这么做；你需要实实在在地画一张状态图。

当然，如果使用某种特定的表述格式是你的业务需求的一部分，你也可以先做决定，然后围绕这种强制的表述格式来设计你的 API。如果你的 API 将成为某个低功耗的嵌入式系统的一部分，你可能必须采用 CoRE Link Format（见第 13 章）来构建 API。如果你所工作的公司是 OData 技术委员会 (OData Technical Committee) 的成员，并且已经部署了 75 个 OData API（见第 10 章），能猜得到——你很可能要设计第 76 号 API。

这并不意味着你的 API 是命中注定的。超媒体为你提供了很多的灵活性。如果你必须那么做的话，你可以将任何你所希望的 API 作为 Maze+XML 的扩展来进行实现。仅仅将它作为 Fielding 约束之上的一个附加约束进行考虑而已。

URL 设计并不重要

一些 API 设计手册，包括最初的《RESTful Web Services》都花了很长篇幅来讨论你要

为资源分配的 URL。你所提供的每个 URL 都应该以一种清楚的方式标识出资源，而人类可以通过这个 URL 来理解 URL 另一端的内容。

如果你发布的资源是一个账户的集合，它应该被称为类似于 `/users/` 的东西。下一级的资源应该发布在父资源的下面。所以表示 Alice 的账号的资源应该使用类似于 `/users/alice` 的 URL。

是的，是这样的。

从技术上而言，这些都不重要。一个 URL 仅仅是某个资源的地址而已，客户端可以使用它来得到一个表述。URL 并不能在技术上说明任何关于资源或者表述的信息。《万维网的架构，第一卷》是这么说的：

人们很容易去检查标识资源的 URI，进而猜测这个资源的性质。但是 Web 是设计用来让 agent 通过表述而非标识符来传递资源信息状态的。一般来说，我们不能通过检查资源的 URI 来判断该资源的表述类型。

这就意味着，对账户的集合采用 URL `/000000000000a` 是完全合法的，而对于该集合中链接到 Alice 的账号的链接可以使用 URL `/prime-numbers?how_many=200`。重要的是，账户集合的表述要清楚地表明它代表的是用户的集合，而 Alice 的账号的表述要包含该资源的状态信息。

当你查看一个 URL 并设法明白它背后的资源的含义时，你就在试图理解这个资源的应用层的语义。这很好。其他书和教程所提到的关于 URL 设计的建议也是很好的建议。但是我不会在本书中重述一遍那些建议，因为我不希望你的 API 依赖于 URL 来传递它们的应用层的语义。我们有更可靠的方式来描述这些内容：媒体类型定义以及机器可读的 profile。

下面是一个例子。当今许多的 API 都在它们的人类可读的资料中记录有 URL 构造规则：一个用户账户的 URL 应该的样式如下：

```
/users/{username}
```

这根本就是一个 URI 模板。如果你提供的表述格式支持 URI 模板，你可以用等价的超媒体控件来代替这份资料。下面是采用 JSON Home Document 格式（见第 10 章）的示例：

```
{
  "user_lookup": {
    "href-template": "/users/{username}",
    "href-vars": {
      "username": "http://alps.io/microformats/hCard#nickname"
    }
  }
}
```


对于一台知道如何解析 JSON Home Document 格式的机器来说，几乎这段代码的所有内容都是机器可读的。唯一需要用人类可读的术语来解释的是语义描述符 `username`。这可以放到内联的文本中，也可以放到一个机器可读的 `profile` 中，比如被链接到这里的 `alps.io profile`。

大部分格式都不正式支持 URI 模板，但是它们包含一个可以做类似事情的超媒体控件——回想一下带有 `action="GET"` 的 HTML `<form>` 标签。你使用这些控件来代替人类可读的文件资料有两大优势。

182 由于超媒体是机器可读的，你的用户可以使用标准库来对其进行管理。这消除了他们误解你的单调的文字说明的可能性。并且由于超媒体控件是运行时提供服务的，你有更大的灵活性来在不破坏客户端的前提下修改这个控件——当这些完全相同的信息是保存在你的 API 资料中时，你是做不到的。

再说一遍，样式美观的 URL 没有任何错误。样式美观的 URL 非常棒！但是它们是装饰品。它们看起来很漂亮。它们什么事也做不了。即便所有的样式美观的 URL 都突然被随机生成的 URL 所代替，你的 API 客户端也应该能够继续工作。

标准名称优于自定义名称

比如说，你的应用语义包含 “a person’s first name”。你在第 1 步时将它记录了下来，然后你试图将它放进一个层级结构中，你也根据你的数据库模式或者你的数据模型中的相对应的字段给它起了一个临时的名字。这个名字差不多类似于 `first_name`、`firstname`、`first-name`、`fn`、`first name`、`first`、`fname` 或者 `given_name`。对于第 1 步而言，这些都是很好的名称。但是当我执行到第三步的时候，你需要往四周看看，你会注意到已经有很多现成的用于描述一个人的名（first name）的 `profile` 了，然后采用其中的一个 `profile`。

你可能更喜欢你在第 1 步中选择的名称。但是你并不是只为你自己设计这个 API。你是在为你的用户设计 API^{注 2}。在他们的职业生涯中，这些用户会使用很多不同的 API，他们会受益于不再需要学习同一事物的 20 种但是区别很小的名称。从短期看，你的用户会受益于不再需要了解你的内部实现细节。你也同样获益：通过为应用语义的常见部分采用标准名称，你可以在不改变 API 的情况下改变内部名称即可。

但是选择哪个 `profile` 呢？hCard 标准描述一个人的 `given name` 的语义描述符是 `given-name`。xCard 标准称为 `given`。而 FOAF 将其称为 `givenName`，但是可以用 `firstName` 来解析旧数据。Schema.org 的 `Person` 项只允许使用 `givenName`。它们都是明确定义的、

注 2 如果你仅仅是为自己设计 API 的话，你可以使用任何名称。

公认的但是又相互冲突的标准。

这有点让人讨厌，但是我们没有理由来编造更多的名称来让情况更糟糕。你只要挑选出从总体而言最符合你的 API 应用语义的 profile，然后使用它所定义的名称即可。

那些负责这些标准的人已经采取措施来避免许多你很可能没有考虑过的概念陷阱。比如，“first name”并不是一个准确的术语。它只是西方文化的一个产物，我们将一个人的名 (given name) 放在前面 (first)。而在某些其他文化中，姓 (family name) 是放在最前面的。中国的现任主席叫作 Xi Jinping。他的“first name”是 Jinping。这也就是 givenname 是一个比 firstname 更好的语义描述符的原因^{注3}。

如果你是一个以英语为母语的人，你很可能不会考虑到这一点。如果你内部的数据库模式有一个称为 firstname 的字段，这也没有关系。但是当你开始将你的数据发送给整个世界的时候，你如何描述这个数据就非常重要了。

hCard、xCard、FOAF 以及 schema.org 的 Person 的设计者都考虑到了这一点。仔细考虑这些棘手的命名问题是他们工作的一部分。这也就是所有这些标准都是用词组 “given name” 来作为它们的语义描述符的基础的原因。这也就是 FOAF 声明 firstName 只能用于解析旧数据的原因。如果你在乎文化敏感性以及文化准确性，你就应该遵循这些现有 profile 的指引。尽可能采用它们，这样你就不再需要那么频繁地处理那些棘手的命名工作。

设计媒体类型

设计新的媒体类型的优点就是你可以完全地控制客户端如何处理你的文档。你不用必须基于 XML、JSON 或者 HTML 来设计你的 API。你可以声明一种全新的二进制文件格式，然后说明如何一个字节一个字节地处理这种文件格式。你不再需要寻找能反映你的应用语义的 profile。一切都是你说了算。

许多组织都拥有仅供内部使用而从不正式声明的文件格式，这些文件格式有的基于 XML，有的基于 JSON。将这些格式变成外部人员可用的、领域特定的超媒体类型并不会花费太多的工作^{注4}。这个工作就是编写正式的规格说明书的工作。一种媒体类型必须提供有完整的、清晰的处理说明书。

如果你总是发现自己在给一个 API 定义 5 种或者 10 种媒体类型，这是一个不好的信号。你应该使用一种通用的超媒体类型来代替，或者你应该定义一种新的媒体类型，加上某

注3 given name 过去通常被称为 “Christian name”：是作为婴儿洗礼仪式的一部分而赠予的名字。这个术语是欧洲天主教文化的产物。并不是世界上的所有人都要经历这一仪式（我就没有），所以我们换成了一个更加通用的术语。现在我们再换了一下而已。

注4 但是将 JSON 格式转换为 Hydra 格式（我将在第 12 章介绍）也并不需要采取太多工作。

些规则来将 ALPS 文档（或者其他 profile 格式）应用到该媒体类型上。那 5 种或者 10 种不同的语义就进入到 5 个或者 10 个 profile 中了。

每种新的媒体类型都需要一个名称，RFC6838 说明了该如何对它们命名。你很可能以一个类似于 `application/vnd.yourcompany.type-name` 的名称而结束。一种基于 JSON 或者 XML 的媒体类型应该使用 `+json` 或者 `+xml` 作为后缀，比如 `vnd.amundsen.application/maze+xml`。

184 如果你期望你所在组织之外的人们能够分发那些使用了你的媒体类型的文件，那么你同样需要告诉他们如何处理这些文档。这就意味着将你的媒体类型注册到 IANA 上。

注册是一个相当正式的过程，它在 RFC6838 的第 4 节和第 5 节进行了描述。大致来说，你需要告诉 IANA（以及所有以后要使用你的媒体类型的人）哪里能找到这个媒体类型的说明书，以及它是否会导致任何特别的安全问题。你可以通过填写表单来注册一种媒体类型。

下面是填写表单时要仔细考虑的主要事情：

- 你必须拥有“一份针对该媒体类型的永久的、现成的、公开的格式规范说明书”。它需要非常详细地对该格式进行说明，并足以让其他人能够仅仅利用规格说明书中的信息就可以为你的数据格式来编写解析器。
你很可能曾经计划将“现成的公开的”媒体类型定义作为你的 API 资料的一部分。但是要达到 IANA 所要求的详细程度所需要的工作量可能远远超过你的预计。这些工作是必要的。因为当你注册了某种媒体类型之后，那些从没听说过你的人、那些和你的 API 没有关系的人都会去访问你的网站，阅读你的规格说明书，这样他们才能使用你定义的格式来生成他们自己的文档。
- 你将需要提及任何与处理你的媒体类型格式的文档所相关的安全问题。如果你的文档能够包含可执行代码的话，这一点是至关重要的。
RFC6838 提供了一个基本的检查清单来思考这个问题。如果你的媒体类型是基于 JSON 的，你还应该参考 RFC 4627 的第 6 节，它描述了和 JSON 相关的安全问题。
- 如果你的媒体类型是基于 XML 的，有一些特别的任务需要你来完成，这些任务在 RFC3023 的 7.1 节中进行了描述。这些任务极有可能包括添加 XML 特定的样板文件到你提交的资料中。
- 如果你的媒体类型不是基于 XML 的，你将需要详细说明这些数据可能会在网络上如何呈现。答案通常是“二进制”。对于基于 JSON 的媒体类型，你可以参考 JSON 标准 RFC 4627，或者简单地说是“二进制”。如果你的媒体类型是基于 XML 的，RFC 3023 的样板文件会负责这部分内容。

- 你很可能应该为你的媒体类型定义 **profile** 参数，这样客户端能够使用 **Content-Type** 报头来请求某种具体的 profile（我在第 8 章中对此进行过讨论）。这是我的观点而已，并不是 RFC6838 的一部分。你可以按照 RFC6906 说的那样，只声明你的媒体类型接收一个 profile 参数，它的值是某个 profile 的 URI。

如果你想要在提交申请之前从社区中获得一些建议，你可以将你已有的内容通过邮件发送至 media-types@iana.org。如果你想查看简单的例子，可以检查一下已经批准注册的 Maze+XML 媒体类型（<http://www.iana.org/assignments/media-types/application/vnd.amundsen.maze+xml>）。

如果你不希望的话，你不需要将你的类型注册到 IANA 上。如果你决定不进行注册，你将需要使用 **vnd.** 前缀（用于商业项目——无论如何，你都可能会使用到的前缀）或者 **prs.** 前缀（用于个人项目或者试验工作）。但是如果你的媒体类型变得更加流行，你真的应该将它进行注册。数以百计的供应商特定（**vendor-specific**）的媒体类型，比如 **application/vnd.ms-powerpoint** 都已经在 IANA 上进行了注册。

当你的API改变时

当今 API 社区中讨论最热烈的一个话题就是版本问题。这是一个严重的问题。大部分公司在发布一个 API 初始版本以后就再也不改变它了。他们不能这么做。

坦率地说，他们不能这么做是由于他们忽略了超媒体限制。大部分的 API 将它们的协议语义和应用语义写进人类可读的资料中。那些 API 的用户之后就基于那些资料编写了一大堆的客户端软件。现在这些 API 提供方进退两难。他们能够修改这些资料，但是这么做并不能自动地改变所有那些客户端的行为。他们赋予了他们的用户对他们任何设计上的改动的否决权。

但是对超媒体文档的修改却能够改变每个收到该文档的客户端的行为。这就是为什么一个网站能够在经过完整的重新设计之后还不会破坏用户的 web 浏览器。网站完全是包含在它所提供的表述中的。这里并不存在额外的隐藏在人类可读的资料中的内容。

问题的关键就在于我在本书中提出的很多建议——这些建议可能最初看起来是学究式的、吹毛求疵的——但是却真真正正地开始获得回报。我的主要目标之一就是减少与 API 相伴的人类可读的资料的数量。这不仅仅是因为人类可读的资料易于被误解，还因为对人类可读的任何修改都需要所有基于这些资料而编写的代码进行相应的改动。

将你的 API 的语义从人类可读的资料移进超媒体文档会让你的 API 面对修改时更加有弹性。选择一种好的超媒体格式，这样你就可以在不影响现有客户端的情况下为你的 API 添加新的资源以及状态转换。你也将有更大的空间来改变你的协议语义。

在理想情况下，对 API 进行重新设计应该像对网站重新设计那样容易。我们很可能永远也不会达到这样的状态，由于同样的原因我们不可能完全地消除语义鸿沟。如果你为状态转换增加了一个新的必需字段，那么你就可以为你的机器可读的 profile 添加一个新的语义描述符，但是这个语义描述符的说明信息还是需要通过人来进行解释。一个完全自动化的客户端也许能够理解它突然收到的错误信息——“你没有为 `required_field` 提供一个值”——但是它不会知道它应该为 `required_field` 发送什么值。

还有一些改动，理想化的客户端能够适应，但是那些现实中的客户端却可能停止工作，比如将超媒体控件从使用 POST 改为使用 PUT。但是总的来说，如果语义是用机器可读的方式进行描述的，你修改语义成功的机会就更多。

如果你修改了一个资源而你的客户端不能自动适应这种变化，你将需要花费一些时间来发布两种不同的资源——旧的资源和新的资源——采用不同的应用语义或者协议语义。这里有 3 种常见的策略来完成这项工作。

URL 空间分割

在最常见的版本控制技术中，整个 API 被分成两套不相交的 API。有时候这两套 API 拥有不同的告示牌 URL，比如 `http://api-v1.example.com/` 和 `http://api-v2.example.com/`。

有时候，只有一个告示牌 URL，但是它的表述使用超媒体来为客户端提供选择版本的机会：

```
<ul>
<li><a class="v1" href="/v1/">Version 1</a></li>
<li><a class="v2" href="/v2/">Version 2</a></li>
</ul>
```

在这里，版本号是一个语义描述符。不知道 v2 含义的客户端不会访问这个链接。

URL 空间分割的方式是起作用的，这是由于 `/v1` 下面发现的表述只能链接到 `/v1` 下面发现的资源。这两个版本的 API 很可能使用相同的底层代码，但是它们可以拥有完全不同的应用语义，因为任何给定的客户端只能排他地使用其中一个。

媒体类型的版本控制

如果你为你的应用定义了一种领域特定的媒体类型，你可以为它提供一个 `version` 参数。客户端之后就可以使用内容协商（content negotiation）（见第 10 章）来请求其中的某个或者另一个版本：

```
Accept: application/vnd.myapi.document?version=2
```

我不认为你应该在一开始就定义一种领域特定的媒体类型，但是即便你这么做了，这也

不是个好主意。你的媒体类型不是你的 API。考虑一下这样的场景：另外一个公司是否能够在他们自己的、与你毫无关系的 API 中使用你的媒体类型呢？除了获得与你的 fiat 标准的兼容性以外，他们还能否从中获得其他益处呢？如果没有令人信服的理由来让其他人采用你的媒体类型，那么就是你将太多属于 API 的东西写进了媒体类型定义中。

你的媒体类型是否像 Maze+XML 那样定义了 API 的协议语义和应用语义的所有方面呢？如果是这样的话，添加一个 `version` 参数是起作用的。按照定义，对语义的修改意味着媒体类型的一次修改。但是，如果你是采用 `profile` 或者任何人类可读的资料而非媒体类型的话，你将很可能在不需要修改媒体类型定义的情况下完成对 API 的修改。这样你会提出一个问题：这个 `version` 属性真正应用到了哪里？媒体类型，还是 API？

标准的媒体类型不会采用这种方式。HTML5 同 HTML4 有很大的不同，但是它们都是作为 `text/html` 来提供的，并且 HTML5 大致基本上向后兼容 HTML4^{注 5}。

profile 的版本控制

我曾建议你采用某种标准化的媒体类型来构建你的 API，这样的话，很明显，你不能进入一个别人的媒体类型中来声明一个新的版本。但是我同样也建议你应用语义定义到一个机器可读的 `profile` 中，而你可以为 `profile` 声明一个新版本。

你的 `profile` 会熟练地将应用中一旦修改就会破坏客户端运行的那部分（因为它们采用人类可读的文本进行描述）从客户端应该能够适应其修改的那部分（因为它们由超媒体进行描述）中独立出来。保留两个 `profile` 可以让你保留两套应用语义。客户端可以使用 `Link` 报头来请求其中的某一个 `profile`。或者，如果媒体类型支持 `profile` 参数的话，客户端可以使用 `Content-Type` 报头来完成正常的内容协商。

版本控制并不特殊

API 版本控制之所以获得了很多的关注是由于忽略了超媒体约束的 API，这个问题特别严重。但是它仅仅是通过超媒体而解决的众多常见问题中的一个例子而已。客户端如何才能知道哪个资源拥有它所想要的表述呢？一旦客户端获得了某个表述，它该如何知道这个表述的含义呢？我前面给你介绍过的技术都是服务器在给客户端提供多个表述以供选择时所通常采用的技术。

服务器可以提供两个链接到不同的 URL 的链接，客户端能够根据对应用语义的理解选择其中一个链接进行访问。不管这两个 URL 指向的是完全不同的资源还是一个资源的

注 5 回溯到 1995 年的 HTML 的 3.0 版本，它和如今的这些 API 所做的事情实际上是相同的。它引入了一个 `version` 参数并建议 HTML 文档应该作为 `text/html;version=3.0` 形式来进行提供。这也被 HTML4 所接受。这样向后兼容性更好。

v1 和 v2 版本，这都是相同的。

188

一个资源可以拥有不同媒体类型的表述。客户端能够使用内容协商（使用 Accept 报头，见第 11 章）或者超媒体来选择它所想要的表述。不论这些媒体类型是完全不同（Collection+JSON 和 HTML）还是只有 version 参数不同，这都是一样的。我认为 version 参数是一个坏主意，但是如果你使用了它，它就跟你使用了两种完全不同的媒体类型一样，也是起作用的。

单个资源可以采用多个不同的 profile 进行描述，而客户端可以使用内容协商或者超媒体来选择它所想要的 profile。不论这些 profile 是对同一概念采用的不同方案（hCard 和 schema.org 的 Person），还是单个 API 的“v1”和“v2”的 profile，这都是相同的。

临终关怀计划

总而言之，版本控制不是一个技术问题。它属于你与你的用户之间的关系的一个方面。你不希望一个小小的改动就破坏所有人的客户端软件，所以你要尽可能多地采用机器可读的超媒体而非人类可读的文档来描述你的改动。当你必须以一种破坏向后兼容性的方式修改某个资源的语义时，你应该创建该资源的第二个版本以供新客户端使用。这第二个版本可以通过一个不同的 URL、不同的媒体类型等来进行标识，没有修改过的客户端可以继续使用老的版本。

最终，你会决定去掉那个老的版本。毕竟，如果你喜欢那个老的功能，你就不会修改它了。需要重复的是，这里并没有什么技术解决方案。这个问题是你与用户的关系问题。你需要为你的 API 什么时候作废、客户端还可以继续使用这个已经作废的 API 多长时间预先做好计划安排。

当你发布你的 API 时，也包含了对该 API 的有效期的一个层次的保证。你可以提出一个使用期保证（“我们将对该 API 提供 5 年的支持”）或者你可以提供一个通知保证（“我们会在停止支持该 API 前保留 1 年的告警期”）。还可以为了沟通方便而特别搭建一个沟通渠道：一个网页或者邮件列表。

当你想要修改你的 API 而这会破坏向后兼容性时，下面这个流程在过去的时间中对我很有帮助：

1. 声明当前版本为“deprecated”。它还是起作用的，但是它已经不再是当前版本了。在你为这个目的而专门搭建的沟通渠道上宣布这个消息。更新你的资料和教程，这样新的开发人员可以以新的版本为起点开始工作，而不再基于那个作废的版本。
2. 经过一段时间后，使用沟通渠道宣布你不再修复“deprecated”的 API 上的 bug 了，并提醒你的用户使用新版本的 API。

3. 经过 no-bug-fix 阶段以后, 宣布那个 “deprecated” 的 API 关闭的最后截止日期。
4. 你很可能需要在截止日期之后有一个宽限期, 但是在截止日期后的某个时间点, 还是要关闭这个旧的 API。所有对旧 API 的请求都会导致 HTTP 状态码 410 (Gone), 而 HTML 的实体消息体来说明这是一个失效的 API 并且链接到新的当前版本。

你多久才能走完这些步骤依赖于你的用户基数的大小以及你的社区可以修改的平均速度。修改一个银行的 API 需要花很长的时间, 而修改一个微博的 API 就快得多了。

听起来不怎么有趣, 是吗? 是的, 很吓人。但是这就是你在不破坏那些已经部署的而你又无法控制的客户端的情况下部署新的服务器软件的过程。这就是超媒体非常重要的原因。你以机器可读的形式记录的协议语义和应用语义越多, 你修改你的 API 而又不需要执行上述费时又费事的流程的能力越强。

不要将所有的超媒体放在一个地方

那些老式的、非 RESTful 的 API 的一个本质特征就是服务描述文档 (*service description document*)。这是一份提供了该 API 的协议语义和应用语义的完整描述的很大的文档 (通常采用 WSDL 格式)。这个文件通常是基于服务器端实现并由理解该 API 的自动化工具生成的。

用户可以下载一份服务描述文档并使用它来自动生成一个对应的客户端实现。他们可以使用这个客户端来像调用本地编程语言调用那样发起远程 API 调用。这个过程并不需要理解诸如超媒体、表述格式或者 HTTP 等任何内容。然后在服务器端实现上某些东西就会发生变化, 你前面所学的东西仿佛一下子都土崩瓦解了。

这种设计的问题就是它造成了 API 的服务器端实现、机器可读描述文档以及从机器可读描述文档中生成的客户端之间的紧耦合。当服务器端实现修改以后, 这些改动不会并反映到那些生成的客户端上, 最终客户端就会停止工作。

现在, 你很可能并没有考虑生成一个你的 API 的 WSDL 描述。但是传统的 API 资料实际上就是一个人类可读的服务描述文档。它是一个描述了你的 API 的应用语义和协议语义的大文件。人类可读的资料比 WSDL 文件更易于让人类理解, 但是它也有同样的问题。服务器端实现的改变会导致“服务文档”的修改, 但是这个修改并不会传导至那些已经部署了的客户端上, 客户端最终就会停止工作。

基于超媒体的 API 有一定的能力来在不破坏客户端的情况下传递服务器端的改变。但是你不会自动获得这种能力; 你需要为此而做一些工作。很可能要以 HTML 来编写一个机器可读的“服务描述文档”。在 Web 上, 我们将其称为站点地图 (site map)。站点地图是一个包含了网站的全部协议语义的完整描述文档。你可以根据一个 HTML 站点地图自

动生成一个 API 客户端。当服务器端实现改变以后，你的客户端就不能正常工作了，因为它所基于的站点地图已经过期了。

超媒体 API 的客户端不能期望预先知道所有可能的状态转换。它需要设计的像迷宫穿越者那样，具有在运行时根据服务器端展示的接下来可能采取的步骤做出决策的能力。这就是我建议将你的超媒体控件进行分割的原因，这样当每个表述被提供给客户端时，其包含的控件是和当前的应用状态以及资源状态相关的。这会迫使客户端开发人员将超媒体考虑在内，而不会装作可以忽略它。

我将其提出是因为有一些自动化工具会检查你的服务器实现然后为你生成一个 API。这个 API 是由一个基于超媒体的服务文档来描述的。从技术上说，这没有错——服务文档中的超媒体依旧是超媒体——但是它会促使你的用户来相信这个服务文档是不会改变的。初看起来，好像一切都很好，但是随着你的 API 的改进演化，你将开始遇到各种问题。你将需要检查标记为“超媒体”的功能，但是你并没有由于采用超媒体而真正获得任何益处。

任何超媒体格式都可以用于编写服务文档，但是有 3 种特别适合作为反模式 (antipattern)。它们是 OData 和 WADL（见第 10 章）还有 Hydra（见第 12 章）。在我介绍这些格式时，我会再次提醒你作为预警。

为现有 API 添加超媒体

假设你已经拥有一个设计并部署了的 API。它是一个具有当今设计的典型特征、fiat 标准的 API，它提供的是不包含超媒体的特定的 (ad hoc) JSON 或者 XML 表述：

```
{
  "name": "Jennifer Gallegos",
  "bday": "1987-08-25"
}
```

你应该能够在不破坏现有客户端的前提下，将你的 API 提升到我在本书中所提倡的质量水准。下面是我对前面介绍过的七步骤流程进行修改后的一个版本，这个版本是用于改进基于 JSON 的 API 的：

1. 将所有已有的表述都记录下来。每个表述都包含一定数量的语义描述符。你不能更改这些描述符，但是你应该能够添加一些新的语义描述符。
2. 为你的 API 画一幅状态图。图中的方框就是你现有的表述。你很可能不拥有任何状态转换，因为大部分现有的 API 都并不拥有任何超媒体链接。现在是时候添加一些了。使用箭头以一种合理的方式来将表述连接起来。箭头的名称就是你的链接关系。

这时候，你的某些语义描述符可能被证明实际上是链接关系：

```
{ "homepage": "http://example.com" }
```

这时候，你可以将它们转换为链接关系，但是要保证不要在下一步修改它们的名称。

3. 你不能修改任何你在第 1 步中写下的内容的名称，因为这可能会破坏你现有的客户端。但是你可以对你在第 2 步中创建的链接关系修改名称，并尽可能保证它们的名称来自于 IANA 以及其他人们所熟知的出处。
4. 你不能更改你的媒体类型，因为这可能会破坏你的客户端。它必须保持使用 `application/json`（或者其他现在所采用的媒体类型）。
5. 由于你不能更改媒体类型，所以你全部的应用语义以及协议语义都必须定义在某些其他的地方。你有两种选择：某个 ALPS profile 或者某个 JSON-LD 上下文。如果你在第 2 步中记下了任何不安全的链接关系，你的最佳选择是采用 Hydra 的 JSON-LD（见第 12 章）。你应该能够将获取到这些 API 调用的人类可读的描述，并将它们转换成机器可读的 Hydra 操作。
6. 你已经完成了大部分代码的编写。你只需要提供合适的链接来扩展每个表述。
7. 你的告示牌 URL 还是跟以前一样。如果你以前并没有这样一个 URL，那是由于你的 API 是由一系列离散的 API 调用组成的，你可以创建一个新的资源来当作你的主页，并且要知道只有理解超媒体的客户端才能够访问它。

改进基于XML的API

这个过程与设计提供 XML 表述的 API 是类似的。你可以使用 XLink 和 XForm（见第 12 章）来为任何 XML 文档添加超媒体控件。

在第 2 步中，当你发现你的某个语义描述符作为链接关系更有意义时，比如下面的 `homepage`：

```
<homepage>http://example.com/</homepage>
```

你不能仅仅将它转换成一个链接关系。这会破坏你现有的客户端。你需要添加一些重复的内容。本例使用 XLink 来将 `homepage` 同时当作链接关系（`xlink:arcrole`）和语义描述符来使用：

```
<homepage xlink:href="http://example.com" xlink:arcrole="homepage">  
  http://example.com/  
</homepage>
```

你可能还会在第 5 步中遇到一些麻烦。你不能在一个 XML 文档中使用 JSON-LD，但是你或许能够编写一个 ALPS profile。如果这些都不能奏效，你可以求助于根据你现有 API 资料而编写人类可读的 profile。

◀ 192

值不值得？

虽然从技术上来说，将一个不支持超媒体的 API 变成一个完整的超媒体 API 是可行的，但是你能从这样的练习中获得的唯一好处可能只是极高的成就感。问题在于，你的旧的 API 已经有客户端了。这就是你不能将其废弃而从零开始设计一个新的 API 的原因。因为现有的客户端并不理解超媒体方面的知识，也就不具有来源于此的灵活性，将它们迁移到新的 API 将是一项非常艰巨的任务。而且为什么你的用户要费心地去学习新的 API 呢？他们已经有能起作用的脚本了。

如果你正计划无论如何都要修改你的 API，使用 profile 和超媒体控件来改进它是有意義的，这样未来的修改会更加容易。但是为现有的 API 添加超媒体并不会解决任何它自己所拥有的问题。

Alice 的第二次探险

在第 1 章中，我曾讨论过一个使用广告牌来推广其 URL 主页的网站。我还讲了一个和 Alice 相关的故事，Alice 这个虚拟的主人公将那个 URL 输入了她的 web 浏览器后，逐步了解了这个网站的功能。

这个故事非常乏味无聊，因为它仅仅是展示了万维网它所应当的工作方式。但是现在我可以讲一个同样类型的但关于 API 的故事，它们之间除了 HTTP 协议以外没有任何共同点。

就跟我之前的故事一样，这个故事也是从一个 URL 开始的，这个 URL 是一个 API 的告示牌 URL：

```
https://www.example.com/
```

（正如你所看到的主机名那样，这个主机名不同于第 1 章的网站以及第 2 章的 API。这个 API 是完全虚构的。）

场景 1：没有意义的表述

这是一个月黑风高的夜晚。一个 HTTP 客户端发起了一个 GET 请求：

193

```
GET / HTTP/1.1  
Host: www.example.com
```

某个人正在操作这个客户端，这个人就是 Alice，第 1 章中的虚拟主人公。但是这一次，她不再使用 web 浏览器，她正在使用一个编程实现的 HTTP 客户端来调查某个新的 API 的功能。没有了 web 浏览器来图形化地展示表述，Alice 可能要费些时间来理解 API 所

做的工作，但是她最终还是能够理解的。

服务器返回一个表述，Alice 对其进行检查，内容如下：

```
200 OK HTTP/1.1
Content-Type: application/vnd.myapi.qbit

===1 wkmje
<{data} {name:"qbe"} 1005>
<{link} {tab:"profile"} "https://www.example.com/The-Metric-System-And-You">
<{link} {tab:"search"} "https://www.example.com/sosuy{?ebddt}">
===2 qmdk
<{link} {tab:"gyth"} "https://www.example.com/click%20here%20for%20prizes">
<{data} {name:"ebddt"} "Zerde">
<{data} {name:"gioi"} "Snup">
```

“这到底是什么东西？”Alice 说道。它既不是 XML 也不是 JSON，里面到处都是像 qbe 这样的没有意义的字符串以及仿佛要把人从 20 世纪 70 年代带到教学幻灯片中遥远年代的链接。

Alice 唯一的线索就是 Content-Type 报头，它标识了某种称为 application/vnd.myapi.qbit 的数据格式，Alice 无处可去了，于是她在 IANA 媒体类型注册表中查找 application/vnd.myapi.qbit。它将她指向了公司网站，这个网站描述了她所正在寻找的这种既非 XML 又非 JSON 的数据格式。这个网站同时还提供了一些用于解析该文件格式的代码库。通过使用这些工具，她能够对她的可编程 HTTP 客户端进行扩展，以使其能够将这个无意义的数​​据流转换为有用的数据结构。

现在 Alice 知道了一些事情。她知道这个文档由两部分组成，一部分称作 wkmje，另一部分称作 qmdk。她也知道了这个文档包含 3 个语义描述符 (gioi、ebddt 和 qbe) 以及 3 个超媒体控件（两个链接和一个 URI 模板）。由于某些奇怪的原因，这个媒体类型将链接关系称为“tab”，这意味着这 3 个超媒体控件有链接关系 profile、search 和 gyth。

但是 Alice 并不知道 wkmje 或者 qmdk 的含义。它们是在媒体类型中进行定义的无意义的单词。其中一个超媒体控件指向 <https://www.example.com/click%20here%20for%20prizes>，但是 Alice 并不知道这个 URL 的另一端是什么，因为这个 URL 看起来像垃圾邮件一样，而且链接关系 (gyth) 也并没有在 IANA 上进行注册。

Alice 知道那个查询控件是一个 URI 模板，它定义了一个叫作 ebddt 的变量，但是她不知道 ebddt 的含义。链接关系 search 是在 IANA 上注册过的，并且通过阅读其定义，Alice 更加相信这是某种类型的查询表单。这意味着 ebddt 很可能是一个查询术语。很有可能有一些东西要对称 ebddt 的语义描述符进行处理，但是 ebddt 是什么意思呢？

194

场景2: Profile

所有这些问题的答案都隐藏在文档的第一个链接背后：

```
<{link} {tab:"profile"} "https://www.example.com/The-Metric-System-And-You">
```

到这时, Alice 已经阅读过本书的第 8 章了。她知道链接关系 `profile` 是在 IANA 上注册过的, 它表示一个链接到某个 `profile` 文档的链接。她发起了她的第二个请求, 并希望能得到一个 `profile` 文档, 这样她就能理解 `ebddt` 和 `gyth` 的含义了：

```
GET /The-Metric-System-And-You HTTP/1.1
Host: www.example.com
```

当 Alice 阅读 `application/vnd.myapi.qbit` 的定义时, 她注意到它包含一些用于将某个 ALPS `profile` 应用到表述的规则, 所以 Alice 希望这是一个 ALPS `profile`。但是即便是一个人类可读的网页也是有用的。

碰巧, 服务器给 Alice 发送的就是一个 ALPS 文档：

```
HTTP/1.1 200 OK
Content-Type: application/vnd.amundsen.alps+xml
```

```
<alps version="1.0">
  <doc>
    可查询的食谱数据库
  </doc>

  <descriptor id="wkmje" type="semantic">
    关于食谱数据库整体的信息
    <descriptor href="#qbe">
    </descriptor>

  <descriptor id="qmdk" type="semantic">
    关于当前特色菜谱的信息
    <descriptor href="#gyth">
    <descriptor href="#ebddt">
    <descriptor href="#gioi">
    </descriptor>

  <descriptor id="qbe" type="semantic">
    表示清单中食谱的总数
  </descriptor>

  <descriptor id="gyth" type="safe">
    链接到食谱的链接
  </descriptor>

  <descriptor id="ebddt" type="semantic">
```

```

    食谱的名称
</descriptor>

<descriptor id="gioi" type="semantic">
    食谱是否符合饮食限制。值 "Snup" 表示素食食谱。"5a" 表示有肉的食谱。其他值也是可以的（比
    如 gluten free、kosher，等等）。但是任何其他值都必须以扩展前缀 "paq-" 开头。如果提供
    了两个或者更多的值，它们必须通过字符 SNOWMAN 来进行分割，比如 "Snup ☃ paq-vegan"
</descriptor>
...
</alps>

```

Alice 可以通过脑力或者自动化工具来将这个 ALPS profile 与 `vnd.myapi.qbit` 文档合并到一起。现在一切都合理了。这个 API 是一个食谱数据库。表述的第一部分将数据库作为整体进行描述。它包括一个根据食谱名称 (`ebddt`) 进行查询的方法以及食谱的总数 (`qbe`)。第二部分是一个指向特色菜谱的链接 (`gyth`)。它提到了菜谱的名称 (`ebddt="Zerde"`)，另外事实上，它是一种素食菜谱 (`gioi="Snup"`)。

将 `vnd.myapi.qbit` 文档与这个 ALPS profile 一起写进一个能够理解这两种媒体类型的程序，这个程序可能会提供如图 9-12 所示的 GUI。

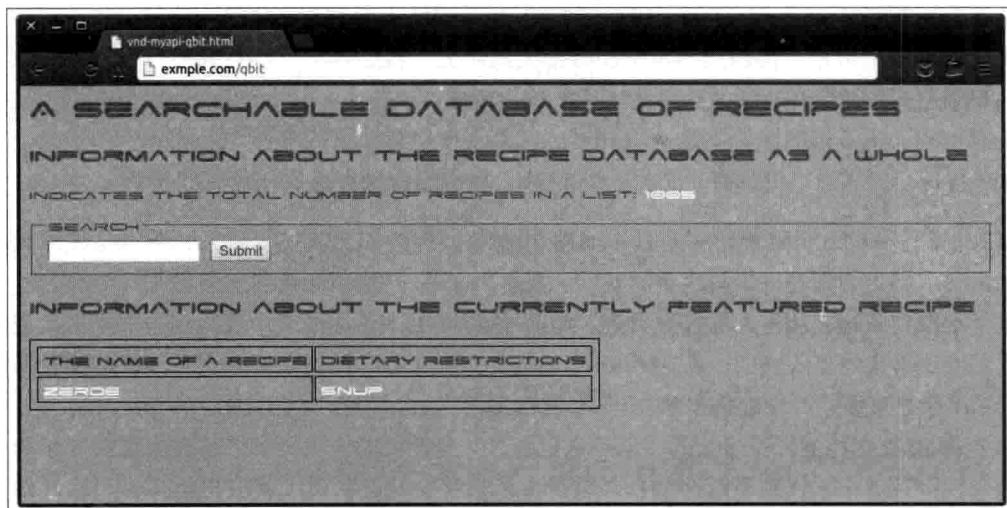


图9-12 采用ALPS profile后可能呈现的qbit界面

这还不够完美——那些人类可读的说明没有被写下来用于这个 GUI 界面，所以这个 GUI 读起来还是很很不方便的——但是就其本身而言，它已经比那个让人费解的 `vnd.myapi.qbit` 文档好很多了。

作为一名程序员，Alice 能够使用 ALPS profile 来实现任何我在图 5-3 中所描述的客户端

类型。下面是一些示例：

- 一个由人类驱动的用于查询食谱数据库的客户端。
- 一个会下载所有它所找到的食谱的爬虫客户端。
- 一个定期运行查询新的素食食谱的监视器。
- 一个能够根据手头的食材来设计一顿饭的代理机器人。这个代理机器人能够利用食谱 API 来找到那些使用现有的食材的食谱。它还会集成一个食品杂货店的报价 API 来查看那些缺少的食材的费用。它的输出就是一份食谱清单，这个清单会使用手头上绝大部分的食材而又拥有最低的额外的费用。

但是碰巧，Alice 一点也不关心做饭的事。在她明白了她最初所收到的文档的含义之后，她就不再使用这个怪异的 API，并且再也不回来了。

Alice明白了

当我设计这个 API 时，我在每一步都尽可能隐藏它的用途。我构造了一种让人迷惑的媒体类型 `application/vnd.myapi.qbit`。我使用一个没有意义的单词 `tab` 来标记链接关系，而非标准术语 `rel`。我所提供的 URL 也是用让人误解的名称来构造的。我使用随机的字母字符串用于给语义描述符和链接关系命名。我还虚构了一些天马行空的、随意的规则用于说明某样食谱是否满足不同的饮食限制。在这个 `vnd.myapi.qbit` 文档中唯一有用的人类可读的字符串是“Zerde”，这是一个特色菜谱的名称^{注6}。此外，由于这个 API 特别通俗易懂：仅仅有一个告示牌 URL，所以也就没有提供任何 API 资料。

尽管如此，Alice 仍然能够了解这个 API 的作用，因为即便我在折磨人，我也是按照我在本书中制定的规则来参与这个游戏的。我提供了一个 `Content-Type` 报头，它包含了这个难以理解的表述格式的媒体类型。Alice 能够在 IANA 注册表中查找到这个媒体类型并阅读该格式的正式说明。在这个 `vnd.myapi.qbit` 文档中，我使用 IANA 注册链接关系 (`search`) 来描述一个查询表单，另一个链接关系 (`profile`) 来链接到一个 `profile` 文档。这个 `profile` 文档是机器可读的，但是它也包含一些重要的人类可读的信息来说明表述的含义。一旦 Alice 找到了这些信息并进行阅读之后，她就明白了其应用语义，然后就知道这不是一个她想要使用的 API。

很明显，你不应该着手给你的用户制造困难。你不应该提供没有意义的 URL，或者随机生成链接关系的名称^{注7}。这个故事要表达的意思是，那些东西在技术层面上都并不重要。你需要确保你的 API 的协议语义和应用语义都通过 `profile` 和媒体类型定义的组合而记录下来了。你需要将你的资料文档作为你的 API 的最重要的一部分，作为一个从其他表述

注6 Zerde 是一款土耳其的餐后甜点，一种大米布丁。我选择它是考虑到并不是太多的读者认识这个单词。

注7 除非你想要绝对确保他们编写的是能够理解超媒体的客户端。

使用超媒体控件和链接关系链接过去的表述来处理，而不要作为一个单独的产品来处理。

在你的 API 表述中，人类可读的链接关系和 URL 都是有帮助的，有提示作用。这些简略的方式可以避免客户端开发人员不断地从你的资料中查找 `ebddt` 这样的名称。这些链接关系和 URL 自己并不是文档资料。那些文档资料是内嵌在你的 API 中的。这才使得你的 API 可以一次次地改变。

超媒体动物园

有很多的超媒体文档格式正在被活跃地使用着，其中的一些是为了某些非常专业的目的而设计的——使用它们的人甚至并没有在意到它们是超媒体格式。而对于其他那些被运用于公共用途的超媒体格式来说，人们也没有真正地考虑过它们。在这一章中，我将会引领你经历一次超媒体“动物园”的教育之旅，在这次游园会中你将会看到最流行和最有趣的超媒体格式。

我不会深入太多的技术细节。这些格式中的任何一员都未必是你想要使用的，但是它们中的很多成员都在本书前面的内容中提到过。而且很多格式都还处于开发阶段，它们的细节很可能还会发生改变。如果你对动物园中所展出的某一个格式感兴趣的话，那么下一步就请去阅读它的正式规范吧。

我的目标是让你对各式各样可以采用的超媒体具备一定的认识，并且向你展示了曾经有多少次我们通过这种认识解决了各种基础问题。这个超媒体动物园中的成员非常齐全，以至于你几乎不需要为你的 API 定义任何全新的媒体类型。你应该可以选择一个现成的媒体类型，然后只要为它编写一份 profile 即可。

我是根据对超媒体介绍的线路来组织超媒体动物园的。其中有一个章节是介绍领域特定格式（根据第 5 章）的，一个章节是介绍以实现集合模式为目标的格式的（根据第 6 章），还有一个章节是介绍通用超媒体格式的（根据第 7 章）。

像 Collection+JSON 这样的格式，我曾经做过非常深入的研究，所以我将会对这些格式做简明的概述，并将你引导回较早的讨论中。有几款超媒体格式我将不会在本章中讨论，因为相较于我在本书中提倡的方式而言，它们采取了不同的方式来实现 REST。我将会在第 12 章谈到 RDF 及其衍生格式，并在第 13 章谈到 CoRE 链接格式。

领域特定格式

这些媒体格式是设计用来在一个特别的领域中描绘问题的。每一种格式都定义了一些非常特定的应用语义，虽然你可以使用它们来传达不同的语义，但是这可能是个坏主意。

Maze+XML

- 媒体类型：application/vnd.amundsen.maze+xml
- 定义方式：个人标准 (<http://amundsen.com/media-types/maze/>)
- 媒介：XML
- 协议语义：使用 GET 链接进行导航
- 应用语义：迷宫游戏
- 涉及章节：第 5 章

Maze+XML 定义了与迷宫、迷宫中的单元格以及单元格之间的连接相关的 XML 标签和链接关系。图 10-1 给出了 Maze+XML 的协议语义的状态图。

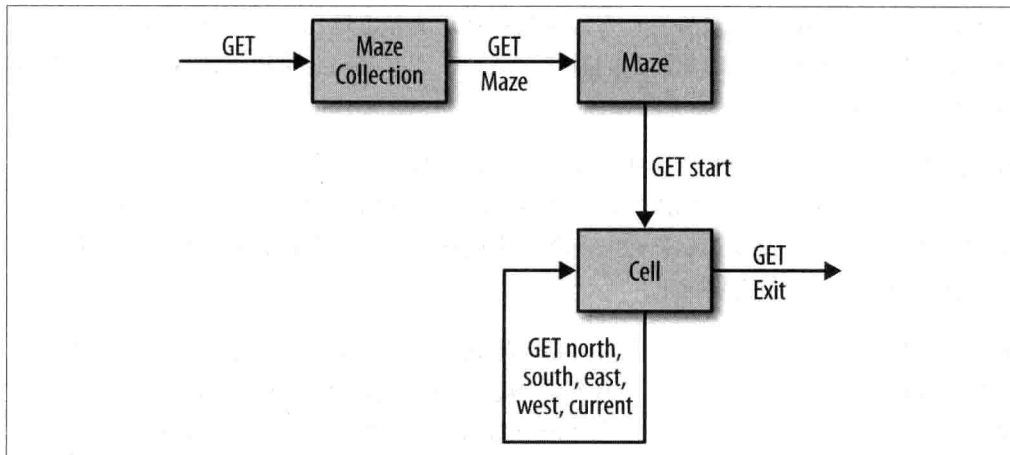


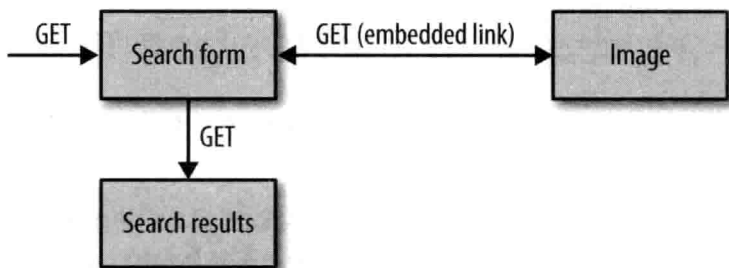
图10-1 Maze+XML的协议语义

Maze+XML 定义了 `<link>` 标签，该标签具有一个链接关系并定义了安全的状态转换；也就是说，它允许客户端发起一个 GET 请求。你可以通过引入自定义的链接关系或定义额外的 XML 标签来扩展 Maze+XML。

我并不真正地推荐使用 Maze+XML，即使你打算制作一个迷宫游戏。它只不过是一个例子，我将它放在第一个介绍的原因是我打算将它作为我展示如何评论超媒体的一个示例。

- 媒体类型：application/opensearchdescription+xml（等待注册中）
- 定义方式：联合标准（consortium standard）（<http://www.opensearch.org/Specifications/OpenSearch/1.1>）
- 媒介：XML
- 协议语义：使用 GET 进行搜索
- 应用语义：搜索查询
- 涉及章节：第 6 章

OpenSearch 是一个用于描述搜索表单的标准。它可以独立使用，或者通过使用 search 链接关系合并到另一个 API 中。它的状态图如下所示：



下面是一段简单的 OpenSearch 表述。OpenSearch 表单的目的地（<Url> 标签的模板属性）是一个与 URI 模板相似的字符串（RFC 6570），虽然它并没有具备 URI 模板的所有功能特性：

```

<?xml version="1.0" encoding="UTF-8"?>
<OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/">
  <ShortName>Name search</ShortName>
  <Description>Search the database by name</Description>
  <Url type="application/atom+xml" rel="results"
    template="http://example.com/search?q={searchTerms}"/>
</OpenSearchDescription>
  
```

OpenSearch 并没有定义表达一次搜索结果的方式。你应该使用符合你的主要表述格式的任何列表格式。

问题细节文档

- 媒体类型：application/api-problem+json
- 描述方式：互联网草案 “draft-nottingham-http-problem”

- 媒介：JSON（具有自动转换成 XML 的规则）
- 协议语义：使用 GET 来进行导航
- 应用语义：错误报告

问题细节文档描述的是错误的状况。它使用结构化的、人类可读的文本来向 HTTP 状态码添加自定义语义。它是一种简单的 JSON 格式，被设计用于替代原本那些你设计出来用以传达错误信息的一次性格式。

就像大多数基于 JSON 的媒体文档，问题细节文档也采用了 JSON 对象的形式。下面是一个跟 503（服务不可用）HTTP 状态码一起发送的文档：

```
{
  "describedBy": "http://example.com/scheduled-maintenance",
  "supportId": "http://example.com/maintenance/outages/20130533",
  "httpStatus": 503
  "title": "The API is down for scheduled maintenance.",
  "detail": "This outage will last from 02:00 until 04:30 UTC."
}
```

这些属性中的两项被定义成了超媒体链接。其中的 `describedBy` 属性是一个链向该表述的人类可读说明的链接。^{注 1}

`supportId` 属性是一个代表了该问题的特定实例的 URL。我们无法预期最终用户会在 URL 的另一端发现什么。它可能会是一个供 API 支持人员使用的内部 URL，又或者可能是一个 URI，一个没有指向任何特定事物的唯一的 ID。

其中 `describedBy` 和 `title` 属性是必需的，而其余的属性是可选的。你也可以为你的 API 添加特定的额外属性。

SVG

- 媒体类型：image/svg+xml
- 媒介：XML
- 协议语义：与 Xlink 类似
- 应用语义：矢量图

SVG 是一种图片格式。它不像 JPEG 代表的是一幅像素级的图片，SVG 是由图形组成的。SVG 包含了一个超媒体控件，可以让一幅图片的不同部分链接到不同的资源。

注 1 `describedBy` 是一个已经在 IANA 注册了的链接关系，它是 `profile` 的一个更加通用的版本。如果一个资源 `describedBy` 任一其他的资源，则说明后者需要对前者的释义进行说明。

上述的超媒体控件是一个 `<a>` 标签，它具有跟 HTML 中的 `<a>` 标签相似的功能。下面是来自第 5 章中的迷宫单元格的一个简单的 SVG 表述：

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">
  <rect x="100" y="80" width="100" height="50" stroke="black" fill="white"/>
  <text x="105" y="105" font-size="10">Foyer of Horrors</text>

  <a xlink:href="/cells/I" xlink:arcrole="http://alps.io/example/maze#north">
    <line x1="150" y1="80" x2="150" y2="40" stroke="black"/>
    <text x="130" y="38" font-size="10">Go North!</text>
  </a>

  <a xlink:href="/cells/O" xlink:arcrole="http://alps.io/example/maze#east">
    <line x1="200" y1="105" x2="240" y2="105" stroke="black"/>
    <text x="240" y="107" font-size="10">Go East!</text>
  </a>

  <a xlink:href="/cells/M" xlink:arcrole="http://alps.io/example/maze#west">
    <line x1="100" y1="105" x2="60" y2="105" stroke="black"/>
    <text x="18" y="107" font-size="10">Go West!</text>
  </a>

</svg>
```

图 10-2 展示了客户端是如何呈现这个文档的。

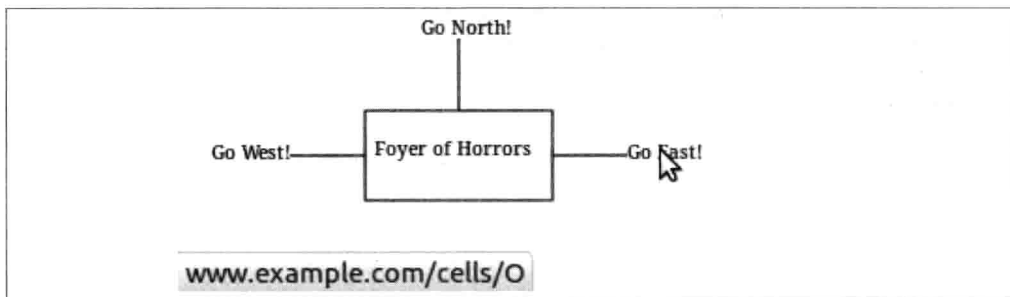


图10-2 迷宫单元格的SVG表述

SVG 成为了除 HTML 之外构建移动应用的另一个很好的选择。SVG 也同样可以与 HTML 5 联合使用：只要在 HTML 标记中使用一个 `<svg>` 标签就可以得到在线的 SVG 图片。

SVG 的 `<a>` 标签实际上并没有定义任何的超媒体能力。它只不过是为 XLink 的 `role` 和 `href` 属性（参见 XLink 规范）准备的占位符标签。因为 SVG 是一种 XML 格式，你也可以向 SVG 添加 XForms 表单（参见 XLink 规范），从而获得堪比 HTML 的协议语

义。不过这并不像向 HTML 嵌入 SVG 那样有用，因为它需要客户端能同时理解 SVG 和 XForms。

VoiceXML

- **媒体类型**：application/voicexml+xml
- **定义方式**：W3C 开放标准 (<http://www.w3.org/TR/voicexml20/>) 及其扩展 (<http://www.w3.org/TR/voicexml21/>)
- **媒介**：XML
- **协议语义**：通过 GET 进行导航；通过表单进行任意的状态转换；GET 用于安全的迁移，POST 用于不安全的迁移
- **应用语义**：口语对话

在第 5 章中，我拿 HTTP 客户端浏览超媒体 API 和人类浏览一棵电话号码的目录树进行了类比。不错，大部分这些电话目录树背后都是以超媒体 API 的方式实现的。他们所使用的表述格式便是 VoiceXML。

下面是第 5 章迷宫游戏中单元格的一份合理的 VoiceXML 表述：

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
    http://www.w3.org/TR/voicexml20/vxml.xsd"
  version="2.1">
  <menu>
    <prompt>
      You are in the Foyer of Horrors. Exits are: <enumerate/>
    </prompt>

    <choice next="/cells/I">
      North
    </choice>

    <choice next="/cells/M">
      East
    </choice>
    <choice next="/cells/O">
      West
    </choice>

    <noinput>Please say one of <enumerate/></noinput>
    <nomatch>You can't go that way. Exits are: <enumerate/></nomatch>
```

```
</menu>
</vxml>
```

205

如果你在手机上玩这个迷宫游戏，你将永远不会直接看到这段表述。而 VoiceXML “浏览器” 存在于手机线路的另一头。当它接收到这段表述时，它会向你大声阅读 `<prompt>` 中的内容：“你现在正处于恐怖大厅。可选择的出口是：north、east 以及 west”，从而以这种方式来处理该文档。

每个 `<choice>` 标签都是一个超媒体链接。浏览器会等你说些什么来激活某个链接。它通过采用语音识别来找出你激活了哪个链接。后面还有一个校验的步骤：如果你什么也没有说或你说了某些无法匹配到任一链接的话时，浏览器便会向你阅读一段错误消息（`<noinput>` 或 `<no match>` 标签中的内容）并等待你再次输入。

一旦你激活了某一链接，浏览器将会向该链接相对应的 `next` 属性所提到的 URL 发起一个 GET 请求。服务器将会响应一段新的 VoiceXML 表述，而浏览器也将会处理这段表述并告诉你现在处在哪个迷宫单元格之中。

`<menu>` 标签仅仅是 VoiceXML 的超媒体控件中最简单的一个。还有一个 `<form>` 标签也使用了语音识别语法，它基于你告诉它的内容来驱动 GET 或 POST 请求。下面是我曾在第 7 章中定义的一个用于触发神奇开关的 VoiceXML 表单：

```
<form id="switches">
  <grammar src="command.grxml" type="application/srgs+xml"/>

  <initial name="start">
    <prompt>
      There is a red switch and a blue switch here. The red switch is
      up and the blue switch is down.

      What would you like to do?
    </prompt>
  </initial>

  <field name="command">
    <prompt>
      Would you like to flip the red switch, flip the blue switch, or
      forget about it?
    </prompt>
  </field>

  <field name="switch">
    <prompt>
      Say the name of a switch.
    </prompt>
  </field>
```



```
<filled>
  <submit next="/cells/I" method="POST" namelist="command switch"/>
</filled>
</form>
```

206 <grammar> 标签是一个与 HTML 的 或 <script> 标签类似的内联链接。它会自动导入一个文档，该文档的格式是由 W3C 的语音识别语法规则（Speech Recognition Grammar Specification）^{注2} 规定的。我不会在这里展示 SRGS 的文件，因为 SRGS 并不是一种超媒体格式。可以这么说，当你说出“触动红色开关”或“忘掉它”时，可以以 SRGS 语法作为依据，从而允许 VoiceXML 浏览器将这些词句转换成一组键值对，这些键值对将匹配表单的 command 和 switch 字段：

```
command=flip
switch=red switch
```

一旦这些字段由通过语音识别获得的值填充之后，<submit> 标签将告诉 VoiceXML 浏览器如何格式化一个 HTTP 的 POST 请求，它看上去就像是一组由 HTML 表单提交的内容：

```
POST /cells/I HTTP/1.1
Content-Type: application/x-www-form-urlencoded

command=flip&switch=red%20switch
```

VoiceXML 文档类似于编程语言的代码。VoiceXML 采用了来自编程的习语，并通过一棵对话树来表示会话流：<goto> 用于从对话的某个部分跳跃到另一部分，<if> 表示的是一个条件，而 <var> 则表示将一个值赋给一个变量。

集合模式的格式

本节中的 3 种标准都具有相似的应用和协议语义，因为它们都实现了我在第 6 章中所展示的集合模式。在集合模式中，某些资源都特指“子项”资源。一个子项通常都会对 GET、PUT 和 DELETE 做出响应，而它的表述都专注在对结构化数据的表现上。其他一些资源则特指“集合”资源，一个集合通常会响应 GET 和 POST-to-append，而它的表述则专注在将子项资源串联起来。

这 3 种标准都采用了不同的方式来实现集合模式；它们可能并不使用术语“collection”或“item”，但是它们都做了几乎相同的事情。

注 2 定义见这里：<http://www.w3.org/TR/speech-grammar/>

Collection+JSON

- 媒体类型：application/vnd.amundsen.collection+json
- 定义方式：个人标准 (<http://amundsen.com/media-types/collection/>)
- 媒介：JSON
- 协议语义：集合模式 (GET/POST/PUT/DELETE)，再加上搜索 (使用 GET)
- 应用语义：集合模式 (“collection” 和 “item”)
- 涉及章节：第 6 章

207

Collection+JSON 被设计作为 Atom Publishing Protocol (参见相关规范) 的基于 JSON 的简单替代选项。它是 API 开发者们在首次经历整个设计流程中非常容易理解的一个正式的、超媒体感知的版本。图 10-3 展示了它的协议语义。

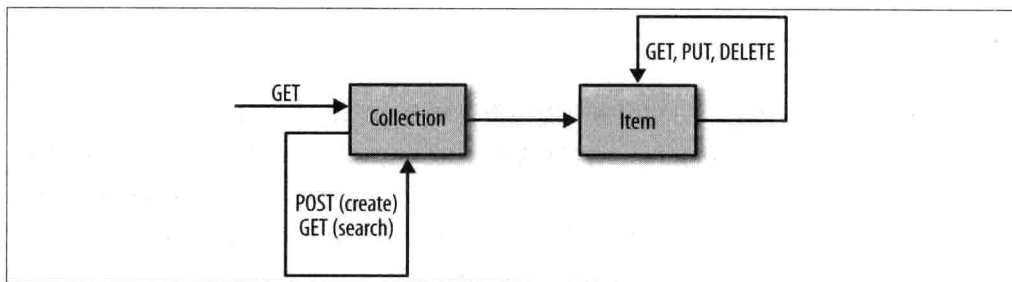


图10-3 Collection+JSON的协议语义

Atom发布协议

- 媒体类型：application/atom+xml、application/atomsvc+xml 和 application/atomcat+xml
- 定义方式：RFC 5023 和 RFC 4287
- 媒介：XML
- 协议语义：集合模式 (GET/POST/PUT/DELETE)；定义良好的扩展为其添加了搜索和其他方式的导航，并全部使用 GET 链接或表单。
- 应用语义：集合模式 (feed 和 entry)；条目具有博客帖子的语义 (作者、标题、类别等)；一个条目并非是一个 Atom 文档 (例如一个二进制图像)，它分成了一个二进制媒体条目和一个包含了元数据的 Atom 条目。
- 涉及章节：第 6 章

原始的 API 标准，AtomPub 在 API 中倡导集合模式和 RESTful 的方式。作为一个基于 XML 的标准，它在这个领域的主宰地位已经被 JSON 表述所取代。AtomPub 现在看来是有点过时了，但是它给多个可以和其他超媒体格式一起使用的标准和链接关系起到

208

了启发的作用，图 10-4 展示它的协议语义。

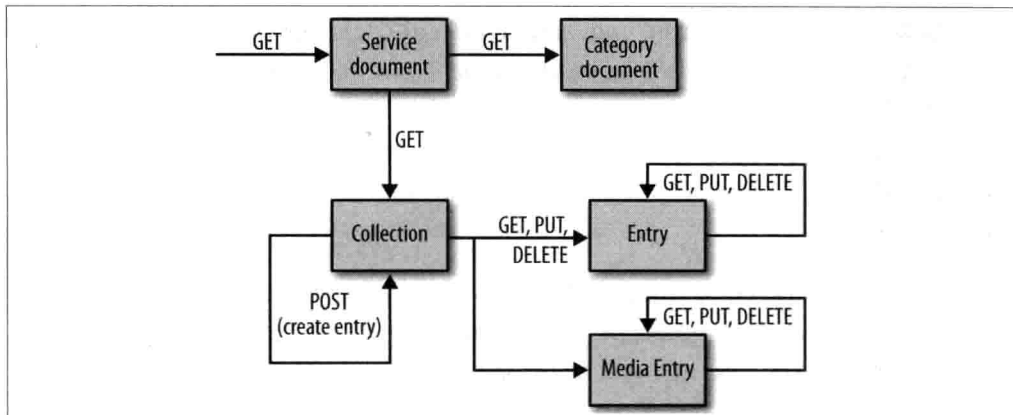


图10-4 AtomPub的协议语义

虽然 Atom 的应用语义暗示了它应该只能被像博客和内容管理 API 这些新闻动态类的应用所使用，但是该标准具有很好的可扩展性。也许最著名的扩展要数谷歌数据协议（Google Data Protocol）了，它是谷歌 API 平台的基础。谷歌通过向 AtomPub 添加领域特定的标签来描述它的每个站点的应用语义。一个 Atom feed 成为了视频的集合（YouTube API）或电子表格单元格的集合（Google Spreadsheets）。

如果你认为你的应用语义不适合集合模式，看看谷歌的 API 目录将可能改变你的想法。谷歌数据协议同样也定义了一个与 AtomPub 的 XML 表述等价的 JSON 版本，不过这只是一个 fiat 标准，不推荐你使用它。

有多个开放标准都定义了 AtomPub 的扩展，包括 Atom 线索扩展（Atom Threading Extensions）和 deleted-entry 元素。我曾在第 6 章讨论过它们。

OData

- 媒体类型：application/json;odata=fullmetadata
- 定义方式：还在制定中的开放标准（<http://www.odata.org/docs>）
- 媒介：一部分是 JSON，其他的是 XML
- 协议语义：修改过的集合模式（GET/POST/PUT/DELETE），支持部分修改的 PATCH 和 GET 查询；通过表单进行任意的状态转换（GET 用于安全的迁移，POST 用于不安全的迁移）
- 应用语义：集合模式（feed 和 entry）

OData 的语义深受 Atom 发布协议的启发。事实上，OData API 可以提供 Atom 表述，而客户端也可以配合一大堆的插件来将 OData API 作为 AtomPub API 使用。我将会把 OData 作为主要提供 JSON 表述的 API 来考虑。图 10-5 展示了 OData 的协议语义视图，简单地展示了我在本节中谈到的部分。下面是一段来自微博 API 的某个集合的 OData 表述，这与第 2 章的 “You Type It, We Post It” 非常相似：

```
{
  "odata.metadata":
    "http://api.example.com/YouTypeItWePostIt.svc/$metadata#Posts",
  "value": [
    {
      "Content": "This is the second post.",
      "Id": 2,
      "PostedAt": "2013-04-30T03:34:12.0992416-05:00",
      "PostedAt@odata.type": "Edm.DateTimeOffset",
      "PostedBy@odata.navigationLinkUrl": "Posts(2)/PostedBy",
      "odata.editLink": "Posts(2)",
      "odata.id": "http://api.example.com/YouTypeItWePostIt.svc/Posts(2)",
      "odata.type": "YouTypeItWePostIt.Post"
    },
    {
      "Content": "This is the first post",
      "Id": 1,
      "PostedAt": "2013-04-30T04:14:53.0992416-05:00",
      "PostedAt@odata.type": "Edm.DateTimeOffset",
      "PostedBy@odata.navigationLinkUrl": "Posts(1)/PostedBy",
      "odata.editLink": "Posts(1)",
      "odata.id": "http://api.example.com/YouTypeItWePostIt.svc/Posts(1)",
      "odata.type": "YouTypeItWePostIt.Post"
    },
    "#Posts.RandomPostForDate": {
      "title": "Get a random post for the given date",
      "target": "Posts/RandomPostForDate"
    }
  ]
}
```

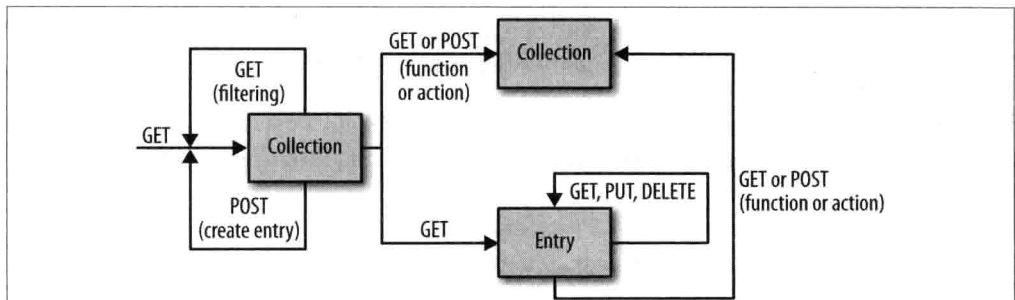


图10-5 OData的协议语义（简化版）

和我们曾经见过的其他基于 JSON 的格式一样，OData 表述也是由一个个 JSON 对象组成的，它们的属性都是由简短、神奇的字符串来命名的。像 `Content` 或 `PostedAt` 这样的属性是普通的 JSON 数据，它们的名字充当了语义描述符。而那些名字中包含了 `odata.` 前缀的属性都是超媒体控件或某些其他 OData 特定的元数据。下面是来自该文档的一些例子：

- 属性 `odata.id` 包含了一个唯一的 ID——这是一个表示特定条目类型的资源的 URI。
- 属性 `PostedAt@odata.type` 包含了 `PostedAt` 属性值的语义类型信息。而类型 `Edm.DateTimeOffset` 引用了 OData 的模式格式：实体数据模型（Entity Data Model）。
- 属性 `odata.editLink` 扮演了类似 AtomPub 中具有属性 `rel="edit"` 的链接。如果你想要修改或删除示例帖子中的某一条，你可以向对应的 URL `Posts(2)` 或 `Posts(1)` 发送 PUT、PATCH 或 DELETE 请求。
- 属性 `PostedBy@odata.navigationLinkUrl` 包含了链向其他资源的超媒体链接。该属性名中应用特定的部分 `PostedBy` 起到了链接关系的作用。站在人类的角度来看，这是链接到发布该帖子的用户的链接。

OData 资源的协议语义重复了你在 `Collection+JSON` 和 AtomPub 所看到过的语义。一个集合资源支持 GET（用于获取表述）和 POST（用于向集合添加新的条目）。条目类型的资源支持 GET，以及 PUT（通过它们的 `odata.editLink`）、DELETE 和 PATCH。

过滤

OData 同样也定义了用于对集合过滤和排序的隐式协议语义，并采用了与 SQL 非常相似的查询语言。如果你知道你得到的是一个链向 OData 集合的 URL，你可以有很多不同的方式来操纵这个 URL。向结果 URL 发送 GET 将可以按不同的方式产出对集合过滤和分页的表述。

我之所以说这些协议语义是隐式的，是因为你无须通过超媒体表单来发起实行特定搜索的 HTTP 请求，你可以基于 OData 规范中的规则来构造这样的请求。

让我们来看一些例子。假设微博集合的（相对）基础 URL 是 `/Posts`。你无须通过超媒体表单来告诉你如何对在 `Content` 属性中包含“second”字符串的博客进行搜索。你可以自行构造 URL^{注 3}：

```
/Posts$filter=substringof('second', Content)
```

注 3 很明显，所有这些 URL 都需要进行 URL 转义。但是为了清晰起见，我都保留了它们尚未转义的形式。

你可以搜索 Content 中含有“second”字样，并且由 Username 属性是“alice”的资源 PostedBy 的博客：

```
/Posts$filter=substringof('second', Content)+ and +PostedBy/Username eq 'alice')
```

你可以只挑选出 2012 年发布的最后 5 条博客：

```
/Posts$filter=year(PostedAt) eq 2012&$top=5
```

想要获取列表的第二页内容吗？你无须在表述中寻找链接关系为 next 的链接，你应该使用的 URL 定义在 OData 规范中：

```
/Posts$filter=year(PostedAt) eq 2012&$top=5&skip=5
```

默认地，微博集合是按 PostedAt 属性值的时间逆序展示条目的。如果你想使用正向的时间顺序来替代，OData 规范说明了你应该使用的 URL：

```
/Posts$orderBy=PostedAt asc
```

在其他的集合模式标准中，服务器必须提供超媒体控件来明确地描述每个允许的搜索分类。Collection+JSON 提供了搜索模板，AtomPub 提供了 OpenSearch 表单。而 OData 集合无须提供这类信息，因为每个 OData 集合都隐式地支持整个 OData 查询协议。客户端不需要通过超媒体表单来了解是否可以向特定的 URL 发送 GET 请求。OData 格式本身在服务器端添加了附加的约束从而保证了特定的 URL 能有效工作。

OData 定义了更多的隐式的协议语义，大部分都与资源之间的关系有关，我在这里就不再赘述了。

函数 (Functions) 和元数据文档

除了在 OData 查询协议中隐式定义的那些令人印象深刻的状态转换集之外，一个 OData 表述还可以包含显式的超媒体控件，用于描述任意的状态转换。这些控件都具有与 HTML 表单相似的语义。我们将安全地转换成为“函数 (functions)”，它们都使用 HTTP 的 GET。将不安全的转换称为“动作 (actions)”，它们都使用 HTTP 的 POST。我将会专注于函数，但是动作也都是以相同的方式工作的。

下面是一个简单的 OData 表单，具有一个日期作为输入。它将会触发一个状态转换，服务器将会从给定日期的全部微博条目中随机选出一条，并将它的表述提供给客户端。

```
"#Posts.RandomPostForDate": {  
  "title": "Get a random post for the given date",  
  "target": "Posts/RandomPostForDate"  
},
```

如果这只是一个例如像“获取指定日期的所有微博条目”这样的简单查询，那么表单将不是必需的。该状态转换将会由 OData 查询协议隐式地描述。但是该协议无法表达“随机选择”这一概念，所以该状态转换必须使用超媒体表单来进行明确的描述。现在，这里便有了一个疑问：你能通过观察这个表单从而指出我们应该发起哪种 HTTP 请求吗？

这个问题实际上是个陷阱。你是无法给出答案的，因为我并没有向你展示整个表单。表单现有的部分是可供你使用的基础 URL (`Posts/RandomPostforDate`)，但是它无法解释如何格式化你所提供的参数，即你想要获得随机博客帖子的日期。它等同于下面这个 HTML 表单：

```
<form action="Posts/RandomPostForDate" method="GET">
  <input class="RandomPostForDate" type="submit"
    value="Get a random post for the given date."/>
</form>
```

这表单明显并不完整，它缺少了对“指定日期”的正式描述。“指定日期”应该是什么格式的？它的语义描述符又是什么？你是应该通过向 `Posts/RandomPostforDate?Date=9/13/2009` 发送 GET 请求来触发状态转换，还是应该向 `Posts/RandomPostForDate?the_date_to_use=13%20August%202009` 发送请求，又或者是向 `Posts/RandomPostForDate?when=yesterday` 发送？你尚未获悉这些信息。

在 HTML 的例子中，缺少的信息应该位于 `<form>` 标签中的第二个 `<input>` 标签中。但是对于 OData 来说，这项信息被保存在另一个文档中——使用 XML 而非 JSON 编写的一个“元数据文档”，并使用了一种叫作逗号模式定义语言 (Comma Schema Definition Language，即 CSDL) 的词汇表^{注4}。

OData 表述使用 `odata.metadata` 属性来链接到它的元数据文档

213

```
{
  "odata.metadata":
    "http://api.example.com/YouTypeItWePostIt.svc/$metadata#Posts",
  ...
}
```

下面是一个元数据文档的一部分，该文档完成了对随机 `PostForDate` 状态转换的定义：

```
<FunctionImport Name="RandomPostforDate" EntitySet="Posts"
  IsBindable="true" m:IsAlwaysBindable="false"
  ReturnType="Post" IsSideEffecting="false">
  <Parameter Name="date" Type="Edm.DateTime" Mode="In" />
</FunctionImport>
```

注4 想要了解更多关于 CSDL 的信息，请访问 OData 的网站 (<http://www.odata.org/documentation/odata-v3-documentation/common-schema-definition-language-csdl/>)。

现在你了解了故事的全部。你通过使用 OData 实体数据模型^{注 5} 定义的格式将日期格式化为字符串，从而触发状态转换为 RandomForDate。你知道这次状态转换是安全的，因为它在 CSDL 描述中的 `IsSideEffecting` 属性被设置为了 `false`。这意味你应该使用 GET 请求来触发状态转换，而不是 POST 请求。

通过将 OData 表述与元数据文档结合，你将可以得到所有必要的信息来触发状态转换 `RandomPostForDate`。你发送了如下的 HTTP 请求：

```
GET /YouTypeItWePostIt.svc/Posts/RandomPostForDate?date=datetime'2009-08-13T12:00' HTTP/1.1
Host: api.example.com
```

虽然 `RandomPostForDate` 是一个简单的转换，但是 OData 状态转换是非常复杂的。元数据文档存储了大量散乱细节，这些细节准确地说明了如何触发你可以从 OData 文档中找到的任意的状态转换。这样便省去了服务器要在每个支持某复杂状态转换的表述中包含该状态转换的完整描述。对该状态转换感兴趣的客户端可以通过元数据文档找到该转换的完整描述。

将元数据文档作为服务描述文档

我采用了某种方式来介绍 OData，从而让它看上去比较像 `Collection+JSON` 或 `Siren`。我们采用包含了像 `DatePublished` 这样的数据字段的 JSON 对象来表现微博，并同时使用超媒体控件和其他“元数据”来说明可能的后续的步骤。

这便是我所推荐的 OData 版本，它具有媒体类型 `application/json;odata=fullmetadata`。但是还可以用另一种方式来编写 OData 文档：这种方式将所有的超媒体控件，都维护在元数据文档中，不仅仅是那些复杂的控件。

214

这种文档的媒体类型是 `application/json;odata=minimalmetadata`。下面是以这种格式生成的微博表述的样式：

```
{
  "odata.metadata":
    "http://api.example.com/YouTypeItWePostIt.svc/$metadata#Posts",
  "value": [
    {
      "Content": "This is the first post.",
      "Id": 1,
      "PostedAt": "2013-04-30T01:42:57.0901805-05:00"
    },
    {
```

注 5 EDM 定义在与 CSDL 相同的文档中。


```

        "Content": "This is the second post.",
        "Id": 2,
        "PostedAt": "2013-04-30T01:45:03.0901805-05:00"
    },
    ]
}

```

这样一来表述瘦身了很多,但是在 REST 的世界里,更小不一定更好。元数据去哪儿了呢? `PostedBy@odata.navigationLinkUrl` 和 `#Posts.RandomPostForDate` 怎么样了? 你如何决定下一步发起哪种 HTTP 请求?

所有的信息都跑到了 `odata.metadata` 链接另一端的 CSDL 文档中。我曾在早前讨论 `RandomPostForDate` 的时候向你展示过部分 CSDL 文档,但是下面的片段添加了更多内容(这段引用的内容展示了 `PostedBy` 和 `RandomPostForDate` 发生了什么情况):

```

<edmx:Edmx Version="1.0"
  xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
  <edmx:DataServices
    xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
    m:DataServiceVersion="3.0" m:MaxDataServiceVersion="3.0">

    <Schema Namespace="YouTypeItWePostIt">
      <EntityType Name="Post">
        <Key><PropertyRef Name="Id"/></Key>
        <Property Name="Id" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Content" Type="Edm.String"/>
        <Property Name="PostedAt" Type="Edm.DateTimeOffset" Nullable="false"/>
        <NavigationProperty Name="PostedBy"
          Relationship="YouTypeItWePostIt.Post_PostedBy"
          ToRole="PostedBy" FromRole="Post"/>
      </EntityType>

      ...

      <EntityContainer Name="YouTypeItWePostItContext"
        m:IsDefaultEntityContainer="true">

        <EntitySet Name="Posts" EntityType="YouTypeItWePostIt.Post"/>

        <FunctionImport Name="RandomPostforDate" EntitySet="Posts"
          IsBindable="true" m:IsAlwaysBindable="false"
          ReturnType="Post" IsSideEffecting="false">
          <Parameter Name="date" Type="Edm.DateTime" Mode="In" />
        </FunctionImport>

        <EntitySet Name="Users" EntityType="YouTypeItWePostIt.User"/>

      </EntityContainer>
    </DataServices>
  </Edmx>

```

```
...  
  
</Schema>  
</edmx:DataServices>  
</edmx:Edmx>
```

在资源的表述之外维护一份该资源的额外信息并没有什么问题。毕竟，这就是 profile 或 JSON-LD 上下文所做的。现在的问题是可以将 CSDL 文档视为一份服务描述文档：将 API 的概述作为一个整体，使它看起来比较像一个关系型数据库。

正如我在第 9 章中所提到过的，看到像这样的文档的用户，会有一种想要基于它自动化生成客户端代码的倾向。这样做会让生成的客户端与服务描述的特定版本建立起紧密的耦合。如果服务器实现一旦发生改变，CSDL 文档将会随着它一起改变。但是客户端无法改变来应对变化，它们将会崩溃。

幸运的是，没有人会让你以这种方式来使用 OData。如果你使用了媒体类型 `application/json;odata=fullmetadata`，你的 OData 表述将会包含它们自有的超媒体控件。客户端在需要触发一个复杂的状态转换（一个函数或动作，不可能由 OData 来完整描述）时只需要查询 CSDL 元数据文档就行了。

纯超媒体格式

这些媒体类型要么就具有非常通用的应用语义，或者就没有任何应用语义，它们专注于表达 HTTP 的协议语义。你可以通过向预定义的属性槽添加链接关系和语义描述符来提供你自有的应用语义。

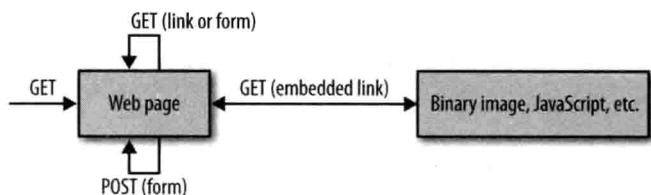
HTML

216

- 媒体类型：text/html 和 application/xhtml+xml
- 定义方式：HTML4 (<http://www.w3.org/TR/html401/>)、XHTML (<http://www.w3.org/TR/xhtml11/>) 和 HTML5 (<http://www.w3.org/TR/html5/>) 的开放标准
- 媒介：类 XML
- 协议语义：通过 GET 链接来进行导航；通过表单进行任意的状态转换（GET 用于安全的转换，而 POST 用于不安全的转换）
- 应用语义：人类可读的文档（“段落”、“列表”、“表格”和“章节”等）
- 涉及章节：第 7 章

HTML 是一种原始的超媒体格式，也是一种在 API 方面被严重低估了的选择。HTML 可以直接使用微格式和微数据，从而替代那种使用例如 ALPS profile 的近似方式。HTML 的 `<script>` 标签让你可以嵌入可执行的代码从而在客户端运行，这是一种其他任何超媒体格式都没有支持的 RESTful 架构特性（“按需代码”，见附录 C）。而且 HTML 文档可以可视化地展示给人类，这对那些被设计用于 Ajax 或移动客户端消费的 API 来说是非常有价值的，在任何种类的 API 的调式过程中也是非常有用的。

下面是 HTML 的状态图：



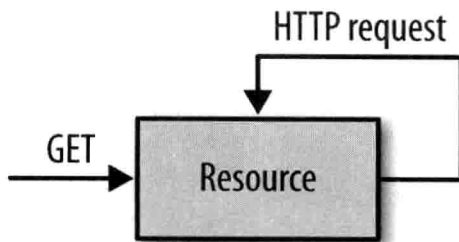
HTML 是由 3 种风味混合而成的。HTML4 自 1997 年开始便已经是一个稳定的标准了。而它的接替者 HTML5 则仍然处于发展之中。其中还包括 XHTML，这是一种符合有效 XML 的类 HTML 格式。

就本书而言，在这 3 种标准中仅有的最重要的区别就是 HTML5 对于客户端输入校验方面的新规则，以及 HTML5 最终支持微数据这一事实。

HAL

- 媒体类型：application/hal+json 和 application/hal+xml
- 定义方式：JSON 版本定义见互联网草案“draft-kelly-json-hal”；XML 版本定义见【个人标准】
- 媒介：XML 或 JSON
- 协议语义：通过可以使用任意 HTTP 方法的链接来触发任意的状态转换；链接中不会提及所使用的 HTTP 方法，这些内容维护在一个人类可读的文档中
- 应用语义：没有谈到
- 涉及章节：第 7 章

HAL 是一种极简主义格式。它的状态图非常通用，看上去很像出自 HTTP 规范：



HAL 依赖于自定义的链接关系（以及 profile 中的人类可读的说明）来完成重要的任务。

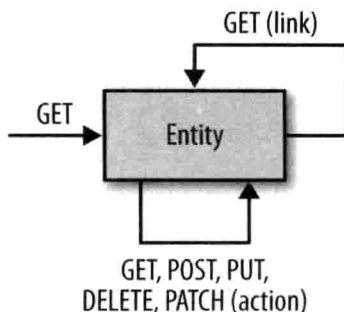
Siren

- 媒体类型：application/vnd.siren+json
- 定义方式：个人标准 (<https://github.com/kevinswiber/siren>)
- 媒介：JSON（XML 的版本尚处于计划中）
- 协议语义：通过 GET 链接进行导航；通过“action”来进行任意的状态转换（GET 用于安全的 action，POST/PUT/DELETE 用于非安全的 action）
- 应用语义：非常通用

一个 Siren 文档描述了一个“entity”，这是一个与 HTML 的 `<div>` 标签具有相似语义的 JSON 对象。一个实体可以具有一个“class”和一组“property”。它可以包含一组“link”，这些链接的工作方式类似于 HTML 的 `<a>` 标签（具有一个 `rel` 和一个 `href` 属性）。它同样还可以包含一组 action，它们的工作方式类似于 HTML 的 `<form>` 标签（具有一个 `name`、一个 `href`、一个 `method` 和数个 `field` 属性）。

一个实体同样也可以具有数个子实体，这与一个 `<div>` 标签可以包含其他的 `<div>` 标签相似。你可以以这种方式来实现集合模式。

Siren 的状态图看上去很像 HAL 和 HTML 的交集：



Link报头

- 媒体类型：n/a
- 描述方式：见 RFC 5988
- 媒介：HTTP 报头
- 协议语义：通过 GET 链接进行导航
- 应用语义：无
- 涉及章节：第 4 章

Link 报头并不是一种文档格式，我将它放在这个动物园里是因为它让你可以将简单的 GET 链接添加到那些缺乏超媒体控件的表述中，比如像二进制图片和 JSON 文档。而报头的 rel 参数是为链接关系而准备的属性槽：

```
Link: <http://www.example.com/story/part2>;rel="next"
```

RFC 5988 为 Link 报头定义了一些有用的参数，包括 type（该参数给了你一些关于链接那头的媒体类型的提示）和 title（包含了链接的人类可读的标题）。

就我而言，Link 报头最重要的作用就是将 JSON 文档连接到一个 profile。JSON 尽管没有超媒体控件却非常流行，而且 application/json 媒体类型也并不支持 profile 参数，所以 Link 便成为了唯一可以指向用于说明 JSON 文档意义的 profile 的可靠方式。

```
Content-Type: application/json
```

```
Link: <http://www.example.com/profiles/hydraulics>;rel="profile"
```

Location和Content-Location报头

- 媒体类型：n/a
- 定义方式：RFC 2616
- 媒介：HTTP 报头
- 协议语义：依赖于 HTTP 响应码
- 应用语义：无
- 涉及章节：第 1 章、第 2 章、第 3 章以及附录

219

下面有两个由 HTTP 标准自身定义的简单的超媒体控件。我曾顺便提到过 Location，但是我会附录 B 对它们进行更细节的介绍。

Content-Location 报头指向了当前资源的权威地址。它等同于使用了已在 IANA 注册的链接关系 canonical 的链接。

每当 HTTP 响应的协议语义需要链接时，Location 报头都会被作为一种通用链接来使用，

而准确的行为取决于 HTTP 状态码。当响应码是 201 (Created) 时, Location 报头指向了一个新创建的资源。但是当响应码是 301 (Moved Permanently) 时, Location 报头指向了一个移动过的资源的新 URL。再次说明, 这些细节将在附录 B 中给予更具体的说明。

URL 列表

- 媒体类型: text/uri-list
- 媒介: 无
- 定义方式: 见 RFC 2483
- 协议语义: 无
- 应用语义: 无

一个 text/uri-list 文档只不过是 URL 的列表:

```
http://example.org/  
https://www.example.com/  
...
```

这可能是曾经设计过的最基本的超媒体类型了。它并不支持链接关系, 所以我们没有办法来表达这些 URL 和提供该列表的资源之间的关系。它也没有显式的超媒体控件, 所以客户端没有办法知道可以向这些 URL 发起怎样的请求。对于你来说最好的办法就是向每个资源发送 GET 请求, 从而看看你所得到的表述是什么类型。

JSON 主文档 (Home Documents)

- 媒体类型: application/json-home
- 定义方式: 互联网草案 “draft-nottingham-json-home”
- 媒介: JSON
- 协议语义: 完全通用
- 应用语义: 无

◀ 220

JSON 主文档是相较于 URL 列表更加复杂的一个版本。这种格式是作为 API 的“主页”来使用的, 它展示了所有 API 提供的资源以及这些资源在 HTTP 协议之下表现的行为。一个 JSON 主文档是一个 JSON 对象。其中的键是链接关系, 而值被认为是“资源对象”的 JSON 对象。下面是一个来自迷宫游戏世界的例子:

```
{  
  "east": { "href": "/cells/N" },  
  "west": { "href": "/cells/L" }
```

```
}
```

资源对象是一个描述了资源协议语义的超媒体控件，也可能是一组相关的资源。下面是一个搜索表单，由 URI 模板描述：

```
{
  "search": {"href-template": "/search{?query}",
             "href-vars": {
               "query" : "http://alps.io/opensearch#searchTerms"
             }
}
```

一个资源对象可能包含了“资源提示”，这种提示可以对协议语义进行更加详尽的描述。最普通的提示是 `allow`，它说明了资源会对哪些 HTTP 方法进行响应。下面的 JSON 主文档使用了我为迷宫游戏的扩展而定义的链接关系 `flip`：

```
{
  "flip": { "href": "/switches/4",
            "hints": { "allow": ["POST"] }
}
```

JSON 主文档对于它所链接的资源的应用语义没有任何描述，相应的信息维护在链接另一端的表述之中。

通过将 JSON 主文档（描述了 API 的协议语义）与 ALPS 文档（描述了 API 的应用语义）结合，你可以选择一个现有的甚至是一个没有使用超媒体的 API，然后将它大部分的人类可读的文档转移到一个结构化的机器可读的格式中。

Link-Template 报头

- 媒体类型：n/a
- 描述方式：互联网草案“draft-nottingham-link-template”（同样见 RFC 6570）
- 媒介：HTTP 报头
- 协议语义：通过 GET 进行导航
- 应用语义：无

Link-Template 报头的工作方式几乎跟 Link 报头是一样的，唯一的差别是它的值是由一个 URI 模板（RFC 6570）来解析的，而非 URL。下面是一个位于 HTTP 报头中的搜索表单：

```
Link-Template: </search{?family-name}>; rel="search"
```

Link-Template 报头具有一个叫作 `var-base` 的特殊变量，它可以让你为 URL 模板中

的变量指定一个 profile。在这个例子中，变量名 `family-name` 是一种提示，它提示了应该向该变量插入什么类型的值，但是从技术上讲它没有任何意义，它也可以叫作 `put-something-here`。通过添加 `var-base`，立马就有了一个指向 `family-name` 正式定义链接。

```
Link-Template: </search{?family-name}>; rel="search"; ↵  
var-base="http://alps.io/microformats/hCard#"
```

现在变量 `family-name` 扩展了 URL `http://alps.io/microformats/hCard#family-name`。该 URL 另一端的 ALPS 文档说明了变量 `variable` 的应用语义。

下面是一个通过使用 `schema.org` 的应用语义来取代 ALPS 的另一个例子：

```
Link-Template: </search{?familyName}>; rel="search"; var-base="http://schema.org/"
```

这里，变量 `familyName` 扩展了 URL `http://schema.org/familyName`，它基本上与 `http://alps.io/microformats/hCard#family-name` 是相同的东西。

在本书撰写之时，互联网草案对于 `Link-Template` 报头的定义已经过期。该草案的作者 Mark Nottingham 让我不管怎样都要将它写入本书。他说如果有更多的人对 `Link-Template` 感兴趣的话，他将会重新发起该草案。

WADL

- 媒体类型：`application/vnd.sun.wadl+xml`
- 定义方式：开放标准 (<http://www.w3.org/Submission/wadl/>)
- 媒介：XML
- 协议语义：完全通用
- 应用语义：无，对扩展有最低程度的支持

◀ 222

WADL 是首个支持整套完整协议语义的超媒体格式。一个 WADL 的 `<request>` 标签（与 HTML 表单类似）可以描述使用任意方法的 HTTP 请求，为任意指定的 HTTP 请求报头提供值，并且包含了任意媒体类型的实体消息体。就像 AtomPub，现在听上去并不觉得它非常特别，但是它曾经是颇具开创性的。WADL 可以描述任意 web API 的协议语义，甚至是那些设计糟糕或违反 HTTP 标准的 API。

下面是一段 WADL 的片段，说明了在第 7 章那版的迷宫游戏中是如何触动开关的：

```
<method id="flip" name="POST" href="/switches/4">  
  <doc>Flip the switch</doc>  
</method>
```


WADL 同样也可以描述 XML 表述的上下文。WADL 文档可以指出表述中那些有意思的部分——尤其是, 哪些部分链接到了其他资源。WADL 文档可以引入一个 XML 模式文档来说明它所描述的 XML 数据的数据类型。当一个 XML 表述没有关联属于它的模式时, 这将会非常有用。

WADL 的 `<doc>` 标签使它可以作为一种基本的 profile 格式, 从而可以对 HTTP 请求的应用语义或 XML 表述的内部进行描述。但是 WADL 却完全无法对 JSON 的内部进行描述。^{注 6}

WADL 并没有得到广泛的使用, 但是有一些 Java JAX-RS 实现可以产生 API 的 WADL 描述。问题的所在是, API 的那些自动产生的描述很可能会与服务端的实现发生紧密耦合。更重要的是, 使用 WADL 的 API 通常会提供一个用于描述整个 API 的协议语义的庞大的 WADL 文档。

这是一个服务描述文档, 我曾在第 9 章有所提及, 它支持用户在基于某种假设时 (即它们所获得的 API 的描述是完整且不变的) 可以创建自动产生的客户端。

但是 API 改变了, 当改变发生时, API 的 WADL 描述也会随着改变, 但是自动产生的客户端不会改变, 它们将会崩溃。

XLink

- 媒体类型: n/a
- 定义方式: W3C 标准 (<http://www.w3.org/TR/xlink11/>)
- 媒介: XML 文档
- 协议语义: 通过 GET 进行导航和引用包含
- 应用语义: 无

XLink 是一个插件标准, 它可以让你为任意的 XML 文档添加超媒体链接。和 HTML 与 Maze+XML 不同的是, XLink 并没有定义那些用于表达超媒体链接的特殊 XML 标签。XLink 定义了一系列属性, 可以应用在任意的 XML 标签中, 从而将该标签转化为链接。

下面是一个属于迷宫游戏中的单元格的特定的 XML 表述。`<root>` 和 `<direction>` 标签是我为了方便示范而创建的标签名, 它们自身并不具有超媒体能力, 但是我们通过向它们添加 XLink 属性从而将它们转化为链接:

```
<?xml version="1.0"?>
<root xmlns:xlink="http://www.w3.org/1999/xlink">
```

注 6 JSON Pointer 标准, 定义见互联网草案 `appsawg-json-pointer`, 可用于处理该问题。

```

<direction
  xlink:href="http://maze-server.com/maze/cell/N"
  xlink:title="Go east!"
  xlink:arcrole="http://alps.io/example/maze/#east"

  xlink:show="replace"
/>

<link
  xlink:href="http://maze-server.com/maze/cell/L"
  xlink:title="Go west!"
  xlink:arcrole="http://alps.io/example/maze/#west"
  xlink:show="replace"
/>
</root>

```

你一定觉得 href 和 title 属性看上去很眼熟，而链接关系被填入了可选的 arcrole 属性。这里稍微有些让人不爽：arcrole 属性只支持扩展的链接关系，即哪种看上去像 URL 的链接关系。你的链接无法像 author 和 east 这种样子，它们必须是像 http://alps.io/maze/#west 这样的。

而 show 属性让你可以在以 HTML 的 <a> 标签 (show="replace", 这是默认的方式) 那样工作的浏览链接和像 HTML 的 标签 (show="embed") 那样工作的嵌入式标签之间进行切换。而所使用的 HTTP 方式则都是 GET。

连同 XLink 一起，我可以给你一个特定的 XML 的词汇表，它所提供的能力与为 Maze+XML 所设计的超媒体能力非常相似。有很多 XLink 的高级特性我并没有提及：尤其是扩展了的链接类型，可以让你使用单个链接将两个以上的资源连接起来；还有 role 属性，我将在第 12 章中进行展示。

XForms

- 媒体类型：n/a
- 媒介：XML 文档
- 协议语义：通过表单进行任意的状态转换（GET 用于安全的转换，POST/PUT/DELETE 用于非安全的转换）
- 应用语义：无

XForms 对超媒体表单的作用就相当于 XLink 对链接所起到的作用。它是一种插件标准，用于为任意的 XML 文档添加类 HTML 的表单。不过，它与 XLink 所不同的是它自身定义了两个标签。下面是一个 XForms 用于表现简单搜索表单的例子：

```
<xforms:model>
```

```

<xforms:submission action="http://example.com/search" method="get"
                    id="submit-button"/>
<xforms:instance>
  <query/>
</xforms:instance>
<xforms:model>

```

`<model>` 标签是一个容器，就好比 HTML 的 `<form>` 标签。`<submission>` 标签对可以发起的 HTTP 请求做了说明：在这个例子中，是一个向 `http://example.com/search` 发起的 GET 请求。`<instance>` 标签的子标签说明了如何构造查询字符串（针对 GET 请求）或实体消息体（针对 POST 或 PUT 请求）。

`<query>` 标签是我为该例子所构造的；它代表了一个被称为 query 的表单字段。我们来举例说明这个标签的意思，在一个 XForms `<input>` 标签中，不管是文本字段还是复选框，都是分开定义的：

```

<xforms:input ref="query">
  <xforms:label>Search terms</xforms>
</xforms:input>

<xforms:submit submission="submit-button">
  <label>Search!</label>
</xforms:submit>

```

具有 `ref="query"` 属性的 `<input>` 标签说明了 query 字段是一个具有人类可读的 `<label>` 的文本输入框。`<submit>` 标签给提交按钮添加了 `<label>` 标签。`<model>` 标签和两个 `<input>` 标签组合在一起的功能等效于下面的 HTML 表单：

```

<form action="http://example.com/search" method="GET">
  <input type="text" name="query"/>
  <label for="query">Search terms</label>
  <submit value="Search!">
</form>

```

225 > 这是一个非常基础的例子，而有很多 XForms 的高级特性我将不会涉及。W3C 的指导手册《XForms for XHTML Authors》^{注7} 使用了 HTML 表单来对 XForms 进行详细说明，并在超越了 HTML 能力的基础之上，对 XForms 的一些高级特性进行了深入介绍。

GeoJSON：一个令人困惑的类型

我们已经在超媒体动物园中看过了那些健康的展览品。现在我要来看看 GeoJSON，这是一个领域特定的文档格式，但是它带有一些缺陷，这些缺陷会伤害 API 的可用性。^{注8} 我

注7 该指导手册见 w3.org 的页面（<http://www.w3.org/MarkUp/Forms/2003/xforms-for-html-authors.html>）

注8 这些缺陷并没有对 GeoJSON 造成很大的影响从而导致无人使用它，它相当流行——它只是没有足够地好。

这样说并不是在挑 GeoJSON 的刺，我自己曾犯过几乎一样的错误，他们是共同的错误，所以即使现在并不需要你去学习 GeoJSON，请牢记它。

GeoJSON 是一个基于 JSON 的标准，被设计用于表示地理特性，比如地图上的点，下面是对它的陈述：

- 媒体类型：application/json
- 定义方式：公司标准 (<http://www.geojson.org/geojson-spec.html>)
- 媒介：JSON
- 协议语义：通过 GET 对坐标系进行引用包含
- 应用语义：地理特性和功能集合

就像大部分在 API 中所使用的基于 JSON 的文档一样，GeoJSON 文档是一个必须包含特定属性的 JSON 对象。下面是一个 GeoJSON 文档，它确定了地球上的某个古老石柱的位置：

```
{
  "type": "FeatureCollection",
  "features":
  [
    {
      "type": "Feature",

      "geometry":
      {
        "type": "Point",
        "coordinates": [12.484281,41.895797]
      },

      "properties":
      {
        "type": null,
        "title": "Column of Trajan",
        "awmc_id": "91644",
        "awmc_link": "http://awmc.unc.edu/api/omnia/91644",
        "pid": "423025",
        "pleiades_link": "http://pleiades.stoa.org/places/423025",
        "description": "Monument to the emperor Marcus Ulpius Traianus"}
      }
    ]
  }
```

226

该表述是由我对现实生活中的 API 做了稍许修改后得到的，该 API UNC 是由古代世界地图中心所提供 (UNC's Ancient World Mapping Center) 的。GeoJSON 的应用语义很简单，而且人类要理解它的文档也是相当容易的。它所表示的集合称为 FeatureCollection。

该集合只包含一个子项：**Feature**，它具有一个几何形状（地图上的单个点）和一连串不同种类的属性，例如人类可读的描述信息。

通过对 GeoJSON 标准的快速浏览，我们发现除了点之外，几何形状也可以是线条（表示边界或道路）或多边形（表示城市或国家的区域）。

GeoJSON没有通用的超媒体控件

不幸的是，GeoJSON 的协议语义一点也不简单。你看到过表述中的 `awmc_link` 和 `pleiades_link` 吗？虽然它们看上去很像超媒体链接，但是它们却不是。根据 GeoJSON 标准，这些都只是正好看上去像 URL 的字符串。当古代世界地图中心在设计它们的 GeoJSON API 时，他们必须将所有的链接填充到 `properties` 列表中，因为 GeoJSON 并没有为他们定义超媒体控件。这意味着一个通用的 GeoJSON 客户端不能访问 `pleiades_link`，又或者根本无法识别出这是一个链接。为了访问该链接，你需要特别为古代世界地图中心的 API 编写客户端。

就算 GeoJSON 没有定义任何的超媒体控件，这也是可以理解的，并不是每种数据格式都会是一种超媒体格式。我原本不会在本书中谈及 GeoJSON，奇怪的是 GeoJSON 确实定义了一个超媒体控件，但是它只能用于一件特殊的事情：改变使用中的坐标系统。

在默认情况下，GeoJSON 表述中的坐标是以经度和纬度来测量的——这是一个我们耳熟能详的系统。由于地球不是一个完美的球体，这些测量值是根据一个叫作 WGS84^{注9} 的标准来解读的，从而能制定出那些与地球形状近似的物体、本初子午线的位置以及“海平面”的概念。

如果你不是一个地图极客，你可以假设地球是一个球体，然后就万事大吉了。但是对于那些地图极客，WGS84 只是默认的，有很多的其他坐标系统可以使用。英国读者很可能对地形测量国家网格 (Ordnance Survey National Grid) 比较熟悉，这是一个采用“easting”和“northing”来代替经度和纬度的坐标系统，但是该系统只能表示覆盖到不列颠群岛 700 ~ 1300 公里区域内的点。我们可以拥有无穷多的坐标系统，因为你可以定义一个系统，然后将诸如本初子午线等任何你想要的地点放入该系统。

现在让我们的故事回到超媒体，因为这就是 GeoJSON 唯一的超媒体控件的作用所在。GeoJSON 让你可以链接到你正在使用中的坐标系统的一段描述。

下面是包含了一个真正的超媒体链接的 GeoJSON 文档，任何的 GeoJSON 客户端都将可以识别这样的文档：

注9 这是一个产业标准，但是与本书提到的其他标准所属的产业都不同。你可以在该页面 (http://earth-info.nga.mil/GandG/publications/tr8350.2/tr8350_2.html) 获取该标准的 PDF 版本。

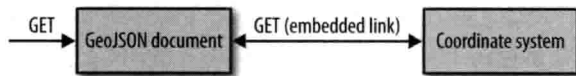
```

{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [60000, 70000]
  },
  "crs": {
    "type": "link",
    "properties": {
      "href": "http://example.org/mygrid.wkt",
      "type": "esriwkt"
    }
  }
}

```

坐标 [60000,70000] 并非有效的经纬度测量值，但是这没有问题，因为我们并不使用经纬度。我们使用的是一个自定义的坐标参考系统（crs），该系统由 <http://example.org/mygrid.wkt> 对应的资源进行表述，这正是超媒体所擅长的。GeoJSON 的问题是唯一可以允许放置链接的地方在坐标参考系统的定义之中。

下面的状态图表述了 GeoJSON 的协议语义：



228

这并不是非常有用！大多数 GeoJSON API 并不使用那些自定义的坐标系统——我们都习惯于普通的经纬度坐标系统。但是 GeoJSON 标准考虑到了它们，因为它们是问题领域的一个重要方面。另一方面，几乎任何 API 都需要在它们的资源之间提供各种各样的链接，但是可能是因为这与问题领域并不直接相关，GeoJSON 标准缺乏这种能力。底层的数据格式也不能带来什么帮助，因为 JSON 完全没有定义超媒体控件。这就是为什么 API 实现者必须凭借像 `awmc_link` 这样的变相方法来达到目的。

抱怨得够多了，那么我可以做些什么不一样的呢？一个更加专注于超媒体的设计将允许这样一组链接，每个链接都可以指定一个链接关系。GeoJSON 更加类似于 Collection+JSON 或 Siren。这样一来，古代世界地图中心便不需要将 `awmc_link` 和 `pleiades_link` 塞进 `properties` 对象了。

为了链接到坐标系统，你将会使用与用于其他任何用途一样的链接类型。GeoJSON 的 `crs` 将会成为一种链接类型，可用于任意的地图应用，即使该应用没有使用 GeoJSON。

拥有应用特定的超媒体控件是没问题的。HTML 的 `` 标签就是一个应用特定的超媒体控件。但是你还提供一个简单通用的链接控件。

GeoJSON没有媒体类型

GeoJSON 还存在另一个问题：它没有已注册的媒体类型。GeoJSON 文档和其他任意的 JSON 文档一样，是以 `application/json` 的格式进行提供的。那么客户端又是如何来区分 GeoJSON 和普通的老式 JSON 的呢？

对于服务器来说最好的解决方案就是将 GeoJSON 作为 JSON 的 profile 对待。这意味着可以采用 `rel="profile"` 的方式向 GeoJSON 标准提供链接。因为 JSON 本身并没有超媒体控件，你将需要使用 Link 报头：

```
Link: <http://www.geojson.org/geojson-spec.html>;rel="profile"
```

你同样也可以为 GeoJSON 编写 ALPS profile 或 JSON-LD 上下文，并使用 Link 报头来提供链接：

```
Link: <http://example.com/geojson.jsonld>;↵  
rel="http://www.w3.org/ns/json-ld#context"
```

据我所知，目前并没有做上述任一工作的 GeoJSON 实现。GeoJSON 以 `application/json` 的媒体类型进行提供，并且我们期望客户端提前知道哪些资源提供的是 GeoJSON 表述，而哪些提供的是普通的 JSON。希望理解不同的 JSON profile 的客户端必须对每个收到的 JSON 表述进行试探，并尝试识别出服务器给它的是哪个 profile。

229

一个客户端需要处理不同的 JSON profile，这听上去是不是有点不切实际？好吧，设想一下，ArcGIS 平台包含了一个可以呈现和 GeoJSON 相同类型信息的 API。它提供了表面上类似于 GeoJSON 表述的 JSON 表述，并将这些表述以 `application/json` 的媒体类型进行提供，并且没有提供 profile 信息。

想象一下有一个客户端可以同时处理 GeoJSON 和 ArcGIS JSON，我并不认为这是一个可笑的幻想。如果以 `application/geo+json` 的媒体类型提供 GeoJSON 表述，而以 `application/vnd.arcgis.api+json` 的媒体类型提供 ArcGIS JSON 表述的话，客户端开发者便可以基于 Content-Type 报头的值来拆分客户端代码，一旦接收到的数据被解析后便可以重新组织代码路径。如果一贯地将 GeoJSON 和 ArcGIS JSON 作为不同的 profile 来提供，开发者便可以基于 Link 报头的值来拆分代码。如果以不同的 JSON-LD 上下文来提供它们的表述，开发者便可以基于上下文来拆分代码。

GeoJSON 和 ArcGIS JSON 这两种格式都会提供，即使它们的意思是一样的。一个统一的客户端必须尝试使用定义糟糕的试探方式来区分这两种格式。又或者，更加有可能的是从来没人思考过统一客户端的想法。就像萍水相逢的路人，各自奔袭在不同的方向上。一个开发者为 GeoJSON API 编写了 GeoJSON 客户端，而另一个则重复了第一个开发者的大部分工作，为了能运行在 ArcGIS 设施上而编写了 ArcGIS 客户端。

没有人会为此追究责任。GeoJSON 标准在 2008 年定稿。回到那个时代，我们对超媒体 API 的理解是非常浅薄的。GeoJSON 的设计者没有忘记注册媒体类型，它们考虑到了这个问题并将问题搁置了起来。

但是现在已经不再是 2008 年了。我们现在已经有可以向 JSON 添加真正的超媒体控件的标准了。我们可以使用 profile 来向通用的超媒体类型添加应用级的语义。我们看到了数以百计的一次性的、互不兼容的以 application/json 媒体类型提供的数据格式，我们知道我们可以做得更好。

从 GeoJSON 学习到的经验

当 GeoJSON 对象被包含在一个具有超媒体能力的 JSON 文档中时（例如 OData 文档，对于内嵌 GeoJSON 有明确的支持），这些问题都将烟消云散。GeoJSON 本身没有通用的超媒体控件，但是这并没有关系，因为它内嵌在一个可以填补这块空白的文档中。GeoJSON 没有特别的媒体类型，但是这也没关系，因为它继承了父文档的媒体类型。在这种情况下，GeoJSON 成为了一种插件标准，类似于 OpenSearch。

如果你要设计一个领域特定的格式，而该格式又不能明确作为某些其他格式的插件，你应该给它定义一个独特的媒体类型。如果你也将该媒体类型注册到 IANA，它便可以对你有所帮助。但是如果你使用 vnd. 前缀，你将不需要注册任何东西。

◀ 230

请确认你的格式具有某些类型的通用超媒体控件，比如 Maze+XML 的 <link> 标签。你可能认为提供通用的超媒体控件并非是你的工作职责，因为这对你的问题领域没有任何帮助。但是如果你不提供超媒体控件的话，你的每个用户都将会鼓捣出他们自己的一次性设计，比如以 awmc_link 的方式。你可以通过为 XML 文档采用 XLink，或为 JSON 文档采用 JSON-LD 来借用某个简单的可插拔的超媒体控件。

总而言之，更好的方式是忘记领域特定的媒体类型，并设计一套领域特定的应用语义——一个 profile。这些语义可以以插件方式添加到某个像 Siren 这样的通用的超媒体类型，或者是一个像 Collection+JSON 这样的集合模式媒体类型。

语义动物园

我已经向你展示了超媒体动物园的奇观，从而展示了基于超媒体设计的多样性和灵活性。现在我将带你去（更加快速地）参观另一个不一样的动物园：一系列充满了不同问题领域的应用语义的蝴蝶花园。我的目标将更加具体：通过复用别人已经完成的工作从而帮你解决时间。

在第9章中，我曾展示了通过复用现存应用语义所带来的好处。这里所陈列的 profile 是那些聪明的人们在对问题领域充分考虑以及经历了棘手的命名问题后所产出的成果。我们没有理由来重复这样的工作，尽可能地复用现存的语义同样可以免去暴露服务器实现细节的不良想法。

最重要的是，当不同的 API 共享相同的应用语义时，编写出可互操作的客户端或通用的语义处理代码库便有了可能性，由此可以改变原来为每个独立的 API 编写自定义客户端的状况。就现在来说，这更多的是一种希望而未变成现实，但是至少眼前前行的道路是明确的。

我不会在语义动物园中向你展示大量独立的 profile，我将主要集中在储藏 profile 的注册中心。

链接关系的IANA注册表

- 媒体类型：任意
- 网站：IANA 页面 (<http://www.iana.org/assignments/link-relations/link-relations.xhtml>)
- 语义：通用导航

231

我谈到过 IANA 注册表，几乎整本书中的链接关系都注册在该注册表。它是一个包含了 60 多个链接关系的全局注册表。允许你在任何表述中使用任何已在 IANA 注册的链接关系，并可以假定你的客户端知道你在表达的内容。

只有链接关系被定义在某个例如 RFC 或 W3C 建议这样的开发标准中，并且足够通用，可以被任意的媒体类型所使用时才可以添加到 IANA 注册表。每个链接关系都拥有一段简短的人类可读的描述，并且拥有一个链向对该关系进行原始定义的标准链接。

在第9章的设计过程的第三步中，我提及过多个对 API 设计非常有用的、已在 IANA 注册过的链接关系。

微格式Wiki

媒体类型：HTML（某些微格式的 ALPS 版本是可用的）

- 网站：微格式页面（<http://microformats.org/wiki>）
- 语义：人类可能想要在线搜索的某类事物

微格式项目是为应用语义定义 profile 的首次成功尝试。微格式是在 wiki 上和邮件列表里以协作方式定义的。就稳定的微格式来说，下面这些可能是你最有可能感兴趣的：

hCalendar

及时地对事件进行描述。基于定义在 RFC 2445 上的普通文本 iCalendar 格式。

hCard

描述了人和机构。基于普通文本的 vCard 格式（定义见 RFC 2426），在第 7 章有所讨论。

XFN

一组用于表述人们之间关系的链接关系，从朋友到同事到爱人。

XOXO

对摘要进行了描述。该微格式非常有意思，因为它没有向 HTML 添加任何东西。它仅仅是对如何使用 HTML 的现存应用语义提出了一些最佳实践的建议。

下面这些微格式规范从严格的意义上来说还都是草案，但是它们中的大部分已经有很多年没有发生变化了，所以我说它们其实都非常稳定：

adr

物理地址。这是 hCard 的子格式，只包含了表示地址的部分。它的思想是指如果你不需要全部的 hCard，你可以只使用 adr。

geo

纬度和经度。（自然而然的是使用 WGS84 标准的）它是 hCard 的另一个子格式。

hAtom

博客文章。基于 Atom feed 格式（RFC 4287）。这是一个超媒体格式（HTML）采用另一个格式（Atom）的应用语义的一个很有趣的例子。

hListing

租赁、个人广告等服务的列表。该微格式通常都会复用相关微格式的语义：
`hReview`、`hCard` 和 `hCalendar`。

hMedia

关于图片、视频和音频文件的基本元数据。

hNews

`hAtom` 的一个扩展，用于向特定的新闻文章添加一些附加的描述符，比如 `dateline`。

hProduct

产品列表。

hRecipe

食谱。

hResume

个人简历 / 简历。

hReview

对（任何事物的）评论的描述，带有评价。

还有一些我尚未提及的微格式也非常有意思，它们是那些可以有效地为 HTML5 所采用，并且现在也已经在 IANA 注册了的链接关系：`author`、`nofollow`、`tag` 和 `license`。而 `rel-payment` 微格式同样也成为了在 IANA 已经注册的连接关系 `payment`。

我已经创建了 ALPS 文档，它囊括了这里所罗列的大部分微格式的必要的应用语义。它们可以通过 ALPS 注册表 (<http://alps.io/>) 获取。

233 来自微格式 Wiki 的链接关系

- 媒体类型：`HTML`
- 网站：微格式页面 (<http://microformats.org/wiki/existing-rel-values>)
- 语义：非常非常地混杂

微格式 `wiki` 同样也具有一个很大的列表，包含了由各种标准定义或在现实用途中见到过

的链接关系，但是它们尚未在 IANA 进行过注册。该 wiki 页面便是记录了可在 HTML5 中使用的链接关系的官方注册表，同样也是所有渴望在单个应用之外能提供使用的链接关系的非官方注册表。Maze+XML 的链接关系从来都不会因为 IANA 而削减，因为它们都太特定于领域了，但是它们都有在微格式 wiki 中有所提及。

在第 8 章中，我曾提到过这个 wiki 页面，并给出了一些在该页面上定义了的关系的例子。我并不建议简单地从该 wiki 页面选取链接关系从而使用它们。你的客户端将会对你所表达的内容毫无头绪。该页面的真正作用是可以作为你查询那些你从未见过的标准的一种途径。

如果你计划开发你自有的迷宫游戏 API，并且你在该页面上搜索了 `maze` 或 `north`，你将会发现 Maze+XML。你不需要就此便开始使用 Maze+XML，但是你可以看看别人是如何解决这种类似的问题的。

schema.org

- 媒介：HTML5 和 RDFa (ALPS 版本可用)
- 网站：schema 主页面 (<http://schema.org/>)
- 语义：人类可能想要在线搜索的某类事物

正如我在第 8 章中所提到的，微数据项的主要来源是一个被称为 schema.org 的交易所。该网站获取像 rNews (针对新闻) 和 GoodRelations (针对在线商店) 这些标准的应用语义，然后将它们适用于微数据项。逐步地，我会自动为 schema.org 的微数据项生成 ALPS 文档，让它们可以在 alps.io 上获取到。

在 schema.org 上有着数百个微数据项的描述，还有更多的尚在制定的进程中，schema.org 的维护者与其他标准的创建者正在为使用微数据来表示这些标准而工作着。我不会讨论所有的微数据项，我将会列出当前顶级的项目并提到它们的一些著名的子类：

- CreativeWork (包括 Article、Blog、Book、Comment、MusicRecording、SoftwareApplication、TVSeries 和 WebPage)
- Event (包括 BusinessEvent、Festival 和 UserInteraction)
- Intangible 是一种全方位的范畴类别，尤其是包括了 Audience、Brand、GeoCoordinates、JobPosting、Language、Offer 和 Quantity
- MedicalEntity (包括了 MedicalCondition、MedicalTest 和 AnatomicalStructure)
- Organization (包括了 Corporation、NGO 和 SportsTeam)
- Person
- Place (包括了 City、Mountain 和 TouristAttraction)
- Product (包括 ProductModel)

234

正如你所看到的，在 schema.org 上的微数据项和微格式之间存在着很多的重叠。Person 子项覆盖到了 hCard 微格式相关的领域。而 Event 子项与 hEvent 非常相似，Article 相对于 hAtom，NewsArticle 相对于 hNews，Recipe 相对于 hRecipe，GeoCoordinates 相对于 geo，等等。

警告：schema.org 的微数据项是非常贴近消费者的。Product 是客户端可以买的东西，但不会是客户端参与其中的工作项目。Restaurant 子项的语义大都是与在餐馆中就餐相关的，并且几乎对项目的运行和检查没有帮助。还有一个 SoftwareApplication 子项，但是它对 bug、单元测试、版本控制仓库、发布里程碑或者我们在开发软件过程中需要处理的任何其他东西都没有关系。就我看来，唯一对从业者有用且描述足够详细的子项是 MedicalEntity，但是某个医生可能会对我的这个观点持不同意见。

简而言之，schema.org 项目有着明确的观点。它并非是百科全书式的，即使它定义的某子项覆盖到了你的 API 领域，它所定义的应用语义可能与你是如何看待事物是没有关系的。

Dublin Core

- 媒介：HTML、XML、RDF 或普通文本
- 网站：Dublin Core 主页 (<http://dublincore.org/>)
- 语义：发布作品

Dublin Core 是用于定义应用语义的原始标准，时间可以追溯至 1995 年。它针对如何发布作品定义了 15 种语义：title、creator、description 等。这些语义可以被作为语义描述符或链接关系来使用。

Dublin Core 元数据启动计划 (Dublin Core Metadata Initiative) 同样也定义了一个更加完整的 profile——DCMI 元数据术语 (DCMI Metadata Terms)。该 profile 包含了像 dateCopyrighted 这样的语义描述符，以及像 isPartOf 和 replaces 这样的链接关系。

235 活动流 (Activity Streams)

- 媒介：Atom、JSON
- 网站：活动流主页 (<http://activitystrea.ms/>)
- 家族：人类在线上做得事情

活动流是一个用于表示我们线上生活的一组连续的离散“活动”的公司标准。每个活动都拥有一个执行者 (actor，通常是某个使用计算机的人)，一个动词 (verb，执行者正在执行的动作) 和一个对象 (object，执行者执行动词的实施对象)。

当你在线观看一个视频的时候，便造就了一个活动。你是执行者，视频是对象，而动词（根据活动流）是字面的字符串“播放”。有一些活动除了对象之外还有一个目标。当我向我的博客发布一条新的帖子时，我就是执行者，博客的帖子是对象，动词是“发布”，而目标就是我的博客。

虽然活动流是一种数据格式，我还是将它安排在了本节，因为数据格式没有定义任何的超媒体控件。不过这里有着大量真正有用的语义。活动流为大部分我们在线上与之交互的事物定义了名称和语义描述符（Article、Event、Group、Person）。更重要的是，它为动词定义了大量有用的名称（join、rsvp-yes、follow、cancel），这让我们通过名字就能识别出不安全的状态转换。

活动流标准说明了如何将一连串活动表示为 Atom feed。采用这种方法，活动流将可以成为一种真正的超媒体格式，一种 Atom 的扩展。

同样也有一个活动流的基于 JSON 的独立版本。它与 GeoJSON 具有一样的问题：没有超媒体控件，而且没有办法区分活动流文档与普通的 JSON 文档。^{注 10} 如果想要向一个 JSON 活动流文档添加超媒体控件，你需要使用 JSON-LD 或 Hydra（第 12 章）。

在活动流的语义和 schema.org 的微数据项之间存在着很多重叠的内容。我们有着叫作 Article、Event、Group 和 Person 的微数据项。UserCheckins 微数据项就类似于活动流中的动词“checkin”，UserLikes 类似于“like”，而 UserPlays 类似于“play”（就记录而言，Activity Streams 早于 schema.org）。

ALPS注册表

为了满足一般的复用需求，我在这个页面（<http://alps.io/>）建立了一个 ALPS profile 的注册表。作为我从媒体类型中解放应用语义这一工作的一部分，我创建了 schema.org 上的元数据项与数个微格式以及 Dublin Core 的 ALPS 版本。这只是一个开始；希望你读到这里时，我将已经完成用于传达其他标准的应用语义的 ALPS profile。

◀ 236

如果你想要使用 ALPS profile 来定义你 API 的应用语义，你可以搜索 alps.io 来找寻可以为你工作的 profile，或者利用几个现有的 profile 来装配一个新的 profile。

如果你打算在你的 API 中使用某个 ALPS profile，可以随意引用 ALPS 注册表中的 profile。一旦你完成了你的工作，如果你能将 profile 上传到 ALPS 注册表，我将会非常感激（同样将它作为你 API 的一部分进行托管）。这样一来，其他人就可以找到并复用你的应用语义了。

注 10 互联网草案“draft-snell-activity-streams-type”可以解决第二个问题。它为活动流文档注册了媒体类型 application/stream+json。

API中的HTTP

让我们将万维网（以及任何其他 RESTful API）看作一个技术架构堆栈。URL 处于最底层；它们用于标识资源。HTTP 协议处于这些资源之上，提供对资源表述的读访问权限以及相应的资源状态的写访问权限。超媒体处于 HTTP 之上，描述了某个特定的网站或者 API 的协议语义。

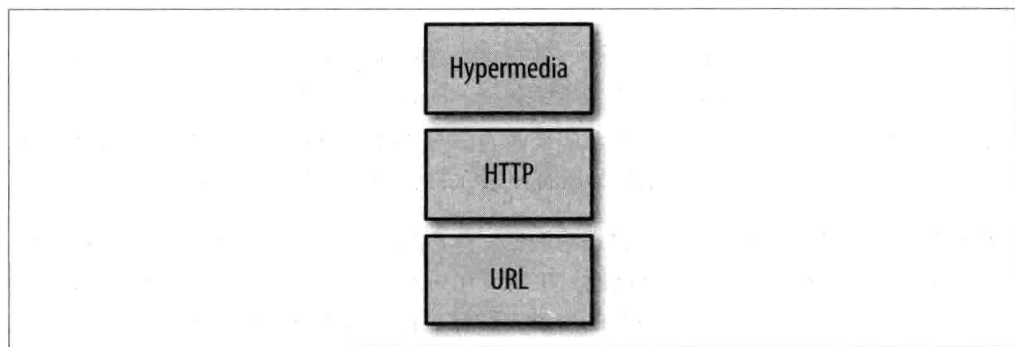


图11-1 万维网的技术架构堆栈

最下面一层回答的是“资源在哪里”的问题，中间那一层回答的是“我如何与这些资源通信”的问题，最上层回答的是“下一步要做什么”的问题。

到目前为止，这本书关注的一直是堆栈的最上面一层——“下一步要做什么”，这是因为最上层这部分内容是最难以处理的。当今大部分的 API 都能正确使用 URL 和 HTTP，但是它们却不能正确使用超媒体，甚至是为超媒体所困扰。

在本章中，超媒体的讲解会告一段落，我会向下一层来讲解 HTTP 的高级协议语义。我不会特别详细地解释 HTTP 协议；如果你需要的话，我建议你阅读由 Gourley 和 Brian Totty 编写的《HTTP: The Definitive Guide》(O’ Reilly)。我会将重点放在那些在 API 中特别有用的 HTTP 特性以及新的 API 开发人员可能不知道的一些 HTTP 特性。

新HTTP/1.1规范

纵观全书，我使用“RFC 2616”来作为 HTTP 1.1 规范的简称。但是 Roy Fielding 和一个 IETF 工作组正在忙于一系列的 RFC 替换的工作，新的 RFC 将代替现有的 RFC 2616。

关于 HTTP 协议的内容没有任何修改；RFC 替换的目的是改进资料文档。新的 RFC 阐明了 HTTP 的协议语义，并合并了一些 RFC 2616 发布之后新定义的扩展内容，比如 *https://* URI 方案的定义。

希望在你阅读这部分内容的时候新的 RFC 已经发布了。但是如果他们的工作还在进行中，你可以访问该工作组的文档清单来阅读那些草案。这比仔细阅读 RFC 2616 更易于理解 HTTP 协议中某些晦涩难懂的部分。

响应码

RFC 2616 定义了 41 个 HTTP 响应码。它们中有一些响应码对我们的目的而言是没有用的，但是总的来说，它们代表了一组基本的语义，而这些语义在最基本的 API 标准中都有定义。我们没有理由忽视这一礼物。如果你为你的 API 重新发明 404 (Not Found) 或者 409 (Conflict) 的话，你这只是在给所有使用你的响应码的人增加负担。

如果客户端向你的 API 发送了一些错误的信息，你应该发送响应码 400 (Bad Request) 以及实体消息体来解释发生了什么问题。不要发送带有关错误消息的 200 (OK)。你那是在欺骗客户端。你将不得不编写另外的资料来解释在你的 API 中：OK 有时候并不意味着“OK”。

在附录 A 中，我讨论了 HTTP 标准定义的所有响应码，以及一些在补充 RFC 中定义的非常有用的响应码。

报头

RFC2616 定义了 47 种 HTTP 请求和响应报头。就和响应码一样，有一些报头几乎没什么用处，但是总的来说，它们定义了一组基本的语义，每个 API 都可以从中获益。务必要使用它们。

有一些报头对应着某些对 API 而言很重要的 HTTP 特性：尤其是内容协商 (content negotiation) 和条件请求 (conditional requests)。我会在本章中对这些特性专门分配章节来进行介绍。在附录 B 中，我讨论了 HTTP 标准所定义的所有报头。另外，我还介绍了一些有用的扩展报头：值得注意的有，你应该已经见过的 Link 报头。

◀ 239

表述选择

单个资源可以拥有多个表述。通常这些表述都采用不同的数据格式：许多 web API 为它们的资源同时提供 XML 和 JSON 风格的表述。有时候，这些表述还包含已经翻译为不同人类语言的文章。有时候这些不同的表述代表了不同的资源状态：一个资源可以拥有一个“概要 (overview) 表述”和一个“详细 (detail) 表述”。

当服务器为一个资源提供多个表述时，客户端应该如何区分它们呢？客户端如何表示它想要的表述是英文还是西班牙文、XML 还是 JSON、概要的还是详细的呢？这里有两种主要的策略。

内容协商 (Content Negotiation)

客户端可以使用特定的 HTTP 请求报头来告诉服务器它想要哪些表述。这个过程就称为内容协商 (content negotiation)，而 HTTP 为此定义了 5 个请求报头。它们统称为 Accept-* 报头。我将在附录 B 中介绍这 5 个报头，但是现在我想要重点强调两个最重要的报头：Accept 和 Accept-Language。

大部分 web API 客户端只理解一种媒体类型。当它们发起一个请求时，它们会发送一个简单的 Accept 报头来请求媒体类型：

```
Accept: application/vnd.collection+json
```

这个客户端告诉服务器，它只理解 Collection+JSON。如果服务器提供 Atom 或者 Collection+JSON 两种选项，它应该提供 Collection+JSON 格式给客户端。

当我使用我的 web 浏览器发起一个 HTTP 请求时，浏览器会发送一个更加复杂的 Accept 报头：

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

RFC 2616 提供了复杂的细节信息来说明哪些内容可以放到 **Accept-*** 报头中，但是在这个来自于现实生活的例子可以作为一个很好的指标来表明哪些内容是可能的选项。Web 浏览器的主要工作是展示网页，所以我的浏览器为 HTML 表述 (**text/html** 媒体类型) 和 XHTML 表述 (**application/xhtml+xml**) 设定了最高优先级。我的浏览器同样可以展示原始的 XML (**application/xml**)，但是由于它看起来并不是很友好，所以 XML 被分配了一个比 HTML 低一些的优先级 ($q=0.9$)。如果 HTML 和 XML 表述都不可用 (可能由于这个资源是一张二进制图片)，我的浏览器会接受任何媒体类型 (**/*/***)。但是这属于下下策，它的优先级也最低 ($q=0.8$)。

240 我的 web 浏览器同样可以设置语言首选项 (language preferences)：我更愿意获取哪种语言的网页。在我发起的每个 HTTP 请求中，我的浏览器会将我的语言首选项配置转换成 **Accept-Language** 报头的一个值而发送进去：

```
Accept-Language: en-us,en;q=0.5
```

这是说，我更愿意接受美式英语的网页，但是我也接受其他英语方言作为次选项 (事实上，我并不真的挑剔这些配置，但是这是我之前所告诉 web 浏览器的内容)。

如果服务器由于 **Accept-*** 限制而不能处理某个请求，它可以发送响应码 406 (**Not Acceptable**)。

Profile 协商

在第 8 章中，我有点反对媒体类型的 **profile** 参数，因为支持该参数的媒体类型并不多，但是它在内容协商中有很大的优势。当某种媒体类型支持 **profile** 参数时，你可以使用内容协商来请求特定的 **profile**。下面内容表示客户端想要获取一个采用了 hCard 微格式的 XHTML 表述：

```
Accept: application/xml+xhtml;profile="http://microformats.org/wiki/hcard"
```

如果 **profile** 已经通过 **Link** 报头传递了，你就不能那么做了。当然，如果媒体类型不支持 **profile** 参数，你也不能这么做。

超媒体菜单

前面所讲的都是内容协商。但是仔细考虑一下通常客户端查找它所想要的表述方式。这种方式中并不存在“资源协商”的过程。相反的是，客户端向 API 的告示牌 URL 发起一个 GET 请求，然后服务器提供一个包含了许多超媒体链接的主页，这些链接都分别链接到其他资源。客户端选择一个它想要访问的链接，之后，就会发起另一个 GET 请求来

获取另一个表述。客户端通过一个接一个地做出选择来找到它所寻找的资源。

当这些选择是在数据格式中做出时，上述策略是同样有效的。HTTP 的内容协商特性很少会针对一些常见的用例进行优化。与使用“内容协商”特性不同，你可以为每个表述分配一个自己的 URL 来使它实际上成为一个独立的资源。

服务器通过发送响应码 300 (Multiple Choices) 来提供在这些资源中进行选择的机会。实体消息体应该包含一个超媒体文档，这个文档链接到不同的选择。如果你这么做，你将需要使用一种能够说明链接的另一端是什么内容的超媒体格式。否则你的客户端就没有办法决定单击哪个链接了。

HTML 的 `<a>` 和 `<link>` 标签通过使用 `type` 属性都可以很好地支持这一要求：

```
<a href="/resource/siren" type="application/vnd.siren+json" rel="alternate">
  The Siren version.
</a>

<a href="/resource/html" type="text/html" rel="alternate">
  The HTML version.
</a>
```

`hreflang` 属性可以表示链接的另一端所采用的语言。

```
<a href="/resource.es" hreflang="es">
  Para la versión en español, haga clic aquí.
</a>
```

因为大部分超媒体格式并不拥有这些功能，我一般建议在这种情况下采用基于报头的内容协商策略。

标准 URL (Canonical URL)

任何时候只要某个资源拥有多个 URL，你就应该将它们中的某一个 URL 标识为正式 URL 或者标准 URL：客户端在提到某个资源（而非向该资源发起请求）时所应该采用的 URL。

这其中有两种方式。第一种就是，你可以将标准的 HTTP 报头 `Content-Location` 作为一个超媒体来指向当前资源的标准 URL。还有一种就是 IANA 注册链接关系 `canonical`，它也可以起到同样的作用。你可以在表述或者 `Link` 报头中使用 `canonical` 链接关系。

HTTP性能

HTTP 客户端被允许在任何时候发起任何它所想要的 HTTP 请求。但是一些请求被证明是对时间没有意义的浪费。HTTP 定义了一些优化方案用于阻拦一些可能没有意义的请求（缓存 (caching)），用于降低某些最终被证明是没有意义的请求的开销（条件请求 (conditional request)），以及用于降低一般的请求的开销（压缩 (compression)）。

缓存 (Caching)

缓存是 HTTP 中最复杂的部分之一。RFC 2616 定义了关于缓存失效的详细规则，这其中还存在许多涉及 HTTP 中间媒介（比如缓存代理）的问题。我将重点讨论一种最简单的方式：使用 HTTP 报头 `Cache-Control`，通过它来为 web API 增加缓存功能。在附录 B 中，我将同样介绍到 `Expires` 报头，它在一些其他常见场景中很有用。对于更复杂的内容，我建议你参考《HTTP: The Definitive Guide》一书和互联网草案“draft-ietf-httpbis-p6-cache”，该草案是当前 RFC2616 替换工作的一部分。

242

下面是一个 `Cache-Control` 报头实例，它是某个对 HTTP GET 请求的响应的一部分：

```
HTTP/1.1 200 OK
Content-Type: text/html
Cache-Control: max-age=3600
...
```

`max-age` 指令是说客户端在再次发起这个 HTTP 请求之前应该等待多长时间。如果客户端获得了这个响应并在半个小时后它想要再次发送这个请求，那么它应该推迟这一操作。服务器已经告诉客户端让它在一个小时（3600 秒）之后回来进行检查，而不是之前。

缓存指令会作用到整个 HTTP 响应上，包括报头和响应码，而不仅仅是实体消息体。其思想是，如果客户端真的需要查看某个响应，它应该查看缓存的响应而非再次发起请求。

`Cache-Control` 的另一种常见的用法是服务器命令客户端不要对响应进行缓存，即便它可以这么做：

```
HTTP/1.1 200 OK
Content-Type: text/html
Cache-Control: no-cache
...
```

这表示，它的资源状态是不稳定的，其表述很可能在被发送到客户端的这段时间中就已经改变了。

当你提供的表述要求你根据本能来判断它的改变频率时，你可以设置 `Cache-Control`

报头。如果你判断错误，这可能导致你的用户使用已经过期的数据。

对于完全由超媒体控件组成的表述以及只有在你升级 API 实现时才会改变的表述，将 `max-age` 设置得比较高是合理的。或者你可以使用……

条件GET请求 (Conditional GET)

有时候，你正好不知道某个资源的状态将要发生改变的时间（集合类型的资源在这方面是最糟糕的）。它可能会一直都在变化，又或者它可能很少变化以至于你都不能估计出其改变的频率。无论哪种方式，你都不能确定 `max-age` 的值，所以你不能要求客户端在一定时间内停止对这个资源发起请求。相反的是，你可以让客户端在任何时候发起它的请求，并在表述没有任何变化时忽略服务器的响应。

这一客户端的功能被称为条件请求 (conditional request)，为了支持这一功能，你将需要在你的表述中提供 `Last-Modified` 或者 `ETag` 报头（同时提供更好）。`Last-Modified` 报头用于告诉客户端该资源的状态上次改变的时间。下面是一个 HTTP 响应的例子：

243

```
HTTP/1.1 200 OK
Content-Length: 41123
Content-type: text/html
Last-Modified: Mon, 21 Jan 2013 09:35:19 GMT
```

```
<html>
...
```

客户端会记录下 `Last-Modified` 的值，并在下次发起请求时，将该值放到 HTTP 报头 `If-Modified-Since` 中：

```
GET /some-resource HTTP/1.1
If-Modified-Since: Mon, 21 Jan 2013 09:35:19 GMT
```

如果资源状态在 `If-Modified-Since` 所指定的日期之后发生了变化，那么不会发生特别的事情。服务器会发送状态码 200、更新后的 `Last-Modified` 以及完整的表述：

```
HTTP/1.1 200 OK
```

```
Content-Length: 44181
Content-type: text/html
Last-Modified: Mon, 27 Jan 2013 07:57:10 GMT
```

```
<html>
...
```

但是如果表述在上次请求之后没有变化，服务器会发送状态码 304 (Not Modified)，并且不再附带实体消息体：

```
HTTP/1.1 304 Not Modified
Content-Length: 0
Last-Modified: Mon, 27 Jan 2013 07:57:10 GMT
```

这节省了双方的时间和带宽。服务器不再需要发送表述，而客户端也不再需要接收它。如果表述是从资源状态中动态生成的，一个条件请求同样节省了服务器生成这个表述的时间。

当然，这意味着你要做一些额外的工作。你将需要跟踪你所有资源的 `last-modified` 日期。你需要记住的是，`Last-Modified` 的值是表述改变的时间。如果你拥有一个集合资源，它的表述包含许多其他的表述，那么这个集合资源的 `Last-Modified` 表示的就是它最后一次子项改变的时间。

244 还有另一种策略比 `Last-Modified` 更易于实现，而且可以避免一些竞争情况。`ETag` 报头（代表“entity tag”）包含一些无意义的字符串，只要对应的表述发生了变化，这个字符串就会相应改变。

如下是一个包含 `ETag` 报头的 HTTP 响应实例：

```
HTTP/1.1 200 OK
Content-Length: 44181
Content-type: text/html
ETag: "7359b7-a37c-45b333d7"
```

```
<html>
```

```
...
```

当客户端对同一个资源发起第二次请求时，它将 `If-None-Match` 报头设置为它在第一次响应中收到的 `ETag`：

```
GET /some-resource HTTP/1.1
If-None-Match: "7359b7-a37c-45b333d7"
```

如果 `If-None-Match` 中的字符串和表述当前的 `ETag` 相同，那么服务器就会发送 304 (`Not Modified`) 和空的实体消息体。如果表述发生了变化，服务器会发送 200 (`OK`)、完整的实体消息体以及更新后的 `ETag`。

提供 `Last-Modified` 需要你记录许多的时间戳（timestamp），但是你不需跟踪任何数据就可以为表述生成 `ETag`。类似于 MD5 散列算法的转换算法可以将任何二进制串转换成一个可靠唯一的短字符串。

问题是，在你运行某种转换算法时，你已经将表述作为二进制串创建出来了。你可以不向网络发送表述来节省带宽，但是你已经完成了构建表述所有必要的工作。如果想要通过使用 `ETag` 来像节省带宽一样节省时间，这就需要你缓存表述的 `ETag`，并在表述

发生变化时使缓存失效。

Last-Modified 或者 ETag 都可以帮助你实现条件请求，但是同时提供这两个报头是更明智的，并且 ETag 比 Last-Modified 更加可靠。

Look-Before-You-Leap请求

条件 GET 请求是设计用来避免服务器向客户端发送大量其已经拥有的表述的。HTTP 的另一个功能，使用不是那么频繁，但是可以避免客户端向服务器发送大量（或者敏感的）表述。对于这种请求并不存在官方的名称，所以《RESTful Web Services》一书为它起了一个土气的名字——look-before-you-leap 请求——这样看起来更容易让人记住。

为了发起一个 LBYL 请求，客户端会发起一个不带有实体消息体的不安全的请求，比如 PUT。客户端同时会将 Expect 请求报头设置为字符串 100-continue。如下就是一个 LBYL 请求的例子：

```
PUT /filestore/myfile.txt HTTP/1.1
Host: example.com
Content-length: 524288000
Expect: 100-continue
```

这不是一个真正的 PUT 请求：它是在询问一个问题，而这个问题是关于将来可能发起的 PUT 请求的。客户端问服务器：“你允许我将一个新的表述 PUT 到 /filestore/myfile.txt 文件吗？”服务器根据这个资源的当前状态以及客户端提供的 HTTP 报头来做出决定。在这个例子中，服务器会检查 Content-Length 并确定它是否愿意接受一个 500MB 大小的文件。

如果回答是 yes，服务器就会发送状态码 100 (Continue)。这样客户端就可以再次发送这个 PUT 请求了，这时要去掉 Expect 报头，并将那个 500 MB 的表述添加到实体消息体中。服务器已经同意接受这个表述了。

如果回答是 no，服务器会发送状态码 417 (Expectation Failed)。回答是 no 或许是因为 /filestore/myfile.txt 上的资源是写保护的，或许是因为客户端没有提供合适的授权证书 (authentication credentials)，或许是因为 500 MB 太大了。不管什么原因，最初的那个 look-before-you-leap 请求已经使得客户端免遭在发送了 500MB 的数据后才被告知数据被拒绝的厄运。这样，客户端和服务端都更好一些。

当然，想要发送一个错误表述的客户端可以通过报头欺骗服务器来得到一个 100 的状态码，但是这没有任何帮助。服务器还是不会接受第二次请求中的那个错误表述的，不会超过它在第一次请求的结果的。客户端大量的数据上传很可能会被响应码 413 (Request

245

Entity Too Large) 所打断。

压缩

像 JSON 和 XML 文档这样的文本表述可以被压缩成它们原始大小的一个零头。HTTP 客户端 lib 库能够请求一个压缩版本的表述并以透明的方式将它进行解压缩提供给它的用户。

它是这样工作的：当客户端发送一个请求时，该请求包含一个 **Accept-Encoding** 报头来说明客户端理解哪种压缩算法。IANA 在它的网页上保存了一份可接受的 **Accept-Encoding** 值的注册表（它是“内容编码 content-codings”清单），但是你想要使用的值是 **gzip**：

```
GET /resource.html HTTP/1.1
Host: www.example.com
Accept-Encoding: gzip
```

246 > 如果服务器理解 **Accept-Encoding** 中提到的某一种压缩算法，它就可以使用该算法来对表述进行压缩，然后再提供给外部。如果表述没有被压缩，服务器就发送和以前相同的 **Content-Type**。但是如果表述被压缩了，服务器就还要发送 **Content-Encoding** 报头，这样客户端才能知道这个文档被压缩了：

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Encoding: gzip
```

[二进制表述在这里]

客户端使用 **Content-Encoding** 所提供的算法来对数据进行解压缩，然后将它按照 **Content-Type** 所提供的媒体类型来进行处理。在本例中，客户端使用 **gzip** 算法来将二进制数据解压缩回一个 HTML 文档。就客户端而言，它请求的是 HTML，收到的也是 HTML。这一技术可以节省大量带宽，而代价却很小，只是增加了一些复杂度。

部分GET请求（Partial GET）

HTTP 部分 GET 请求允许客户端来只获取表述的一部分子集。它通常用于恢复以前被中断的下载。大部分 web 服务器支持对静态内容（static content）的部分 GET 请求。如果你的 API 提供很大的静态文件，花些工夫来为它们实现部分 GET 请求是值得的。

当某个资源支持部分 GET 请求时，它会在一个正常 GET 请求的响应中将 **Accept-Ranges** 响应报头设置为一个字符串“bytes”以突出这一特性。假设有一个 GET 请求

去获取一个非常大的视频文件，下面就是对该请求的响应：

```
HTTP/1.1 200 OK
Content-Length: 1271174395
Accept-Ranges: bytes
Content-Type: video/mpeg
```

[二进制表述在这里。]

如果下载被中断了，支持部分 GET 请求的客户端可以从被中断的地方恢复下载，而非从头开始。下面的请求用于获取上述视频文件最后 1KB：

```
GET /large-video-file
Range: 1271173371-
```

响应应该如下：

```
206 Partial Content
content-Type: video/mpeg
Content-Range: 1271173371-1271174395
Content-Length: 1024
```

[二进制表述在这里。]

247

从理论上说，部分 GET 请求可以用于将一个表述按照逻辑模块进行切分，而不是简单地分割为比特数据块。在这种情况下，**Accept-Range** 报头就需要有另外一个值而非“bytes”了，而 **Range** 报头也就用于获取那些逻辑模块了，比如说，获取某个有 5 个子项的集合的第二个子项。

这个思想非常好，但是在这个领域还没有任何标准发布，一般，我反对创建你自己的协议语义。如果你想要将一个集合进行切分，以便于通过多个 HTTP 请求来获取整个集合，你应该创建一些“page”资源，并使用 IANA 注册链接关系，比如 **next** 和 **previous** 来将它们的表述链接到一起。

Pipelining

Pipelining 技术通过允许客户端一次发送多个 HTTP 请求来降低延迟。服务器按照它收到请求的顺序来将响应发送回去。Pipelining 技术依赖但又不同于持久化连接，持久化连接是一种让客户端通过单个 TCP 连接来发送多个请求的 HTTP 功能。

在 Pipelining 技术中，客户端可以将一系列幂等的 HTTP 请求放入管道中，前提是这些请求作为整体同样是幂等的。如果连接被中断的话，你必须能够重新发起整个请求序列并得到相同的结果。

如下是一个简单的例子。我将在一个 HTTP 管道中发送两条请求。首先我将获取某个资源的表述，然后我会删除这个资源：

```
GET /resource
DELETE /resource
```

GET 和 DELETE 都是幂等的，但是它们合并以后就不是了。如果在我发送这些请求后网络出现了问题，而我也并没有从管道中收到响应的话，那么我将不能再次发送这些请求并得到相同的结果。这个资源已经消失了。正是由于这种复杂性，所以我只建议你对一系列 GET 请求采用 pipelining 技术。

除了上述的复杂性以外，Pipelining 技术并不能帮助提高性能。只有当客户端对同一个域名发起非常多的 HTTP 请求，Pipelining 才是值得的，而大部分网站的各个组成部分都来自于不同的域名。

非浏览器的 API 客户端常常对单个域名发起一长串的请求，但是 Pipelining 对于基于超媒体的 API 并不是特别有用，因为超媒体 API 通常需要客户端在发起新的请求前先查看当前请求所收到的响应。或许，这就是大部分可编程 HTTP 客户端 lib 库不支持 Pipelining 的原因。

248 从根本上说，这个功能并不怎么好。HTTP 2.0 协议（请见本章最后一节）应该以一种更有用的方式来实现 HTTP Pipelining 功能。像它现在这样的状况，Pipelining 对于第 5 章的地图生成器的客户端或者运行在具有高时延的手机设备上的客户端还是可以有用的。它并不像部分 GET 请求那样是必需品，但是当你考虑提升性能时，Pipelining 也是值得考虑的。这是我所能给你的最好的建议。

避免更新丢失问题

我将 ETag 和 Last-Modified 作为一种发起 GET 请求时用于节省时间和带宽的方法。但是在使用诸如 PUT 和 PATCH 等不安全的 HTTP 方法时，条件请求 (conditional requests) 作为一种避免数据丢失的方法也是同样有用的。

假设 Alice 和 Bob 正在使用不同的 API 客户端来编辑一个食品杂货店的物品清单。他们发起了完全相同的 HTTP 请求：

```
GET /groceries HTTP/1.1
Host: www.example.com
```

然后收到了相同的表述：

```
HTTP/1.1 200 OK
```

```
Content-Type: text/plain
ETag: "7359b7-a37c-45b333d7"
Last-Modified: Mon, 27 Jan 2013 07:57:10 GMT
```

```
Pastrami
Sauerkraut
Bagels
```

Alice 向货物清单中添加了一项内容，然后将这个新的表述 PUT 回去：

```
PUT /groceries HTTP/1.1
Host: www.example.com
Content-Type: text/plain
```

```
Pastrami
Sauerkraut
Bagels
Eggs
```

她收到了一个带有 200 (OK) 字样的响应。

Bob，对 Alice 的行为并不知情，他向货物清单中添加了一项内容并将这个新的表述 PUT 回去：

```
PUT /groceries HTTP/1.1
Host: www.example.com
Content-Type: text/plain
```

```
Pastrami
Sauerkraut
Bagels
Milk
```

249

Bob 同样得到了一个响应码为 200 (OK) 的响应。但是 Alice 版本的货物清单——那个包含“Eggs”的清单版本——已经丢失了。Bob 甚至都不知道曾经出现过那样一个版本的清单。

这种悲剧是可以通过发起不安全的条件请求 (conditional) 请求而避免的。当我们希望 GET 请求只有在表述发生了变化以后才会顺利获取到新的表述时，可以使用条件 GET 请求。现在 Bob 想要他的 PUT 请求只有在表述没有发生变化的情况下才会顺利执行。所需要的技术是相同的，但是这个条件现在颠倒了。客户端不再使用 `If-Match`，而是使用一个相反的报头 `If-None-Match`。客户端也不再使用 `If-Modified-Since`，而是使用 `If-Unmodified-Since`。

假设 Bob 发起了他的 PUT 条件请求：

```
PUT /groceries HTTP/1.1
Host: www.example.com
Content-Type: text/plain
If-Match: "7359b7-a37c-45b333d7"
If-Unmodified-Since: Mon, 27 Jan 2013 07:57:10 GMT
```

```
Pastrami
Sauerkraut
Bagels
Milk
```

服务器这一次发送的状态码是 412 (Precondition Failed), 而不再是 200 (OK)。Bob 的客户端也就会知道其他人已经修改过这个货物清单。Bob 的客户端就不会覆盖当前的表述, 而是可以再次发送一个 GET 请求来获取新的表述, 并将它与 Bob 之前的货物清单进行合并。或者它也可以将这个问题进行上报, 由 Bob 本人来处理这个问题。这取决于媒体类型和应用程序。

在我看来, 你的 API 实现应该要求客户端发起的必须是条件 PUT 和 PATCH 请求。如果客户端试图发起一个不带条件的 PUT 或者 PATCH 请求, 你应该发送状态码 428 (Precondition Required)。

认证

为简单起见, 我在本书中所展示的例子都不要求任何类型的认证。你发起一个 HTTP 请求, 就会收到一个响应。这样的 API 在现实中有许多, 但是大部分的 API 都要求认证。

认证有两个步骤。第 1 步是一个一次性的步骤, 用户通过服务提供方设置自己的证书。通常, 这意味着一个人要使用他的 web 浏览器来在 API 服务器上创建一个账号, 或者将某个网站上现有的账号与这个 API 服务器进行绑定。

250 第 2 步是在每次向 API 发起请求时自动提交用户证书。

为什么要在每次 HTTP 请求中都提交用户证书呢? 这是由于无状态约束, 无状态约束要求服务器完全忽略请求中的客户端。在 RESTful 服务器实现中并不存在会话(session)^{注 1}。

有一些认证技术还包括一个称为注册的“第 0 步”。在这一步中, 开发人员要使用他的 web 浏览器为他所正在编写的软件客户端设置证书。如果有一千个人使用这个客户端的话, 他们每个人都必须设置他们自己的个人用户证书(personal user credentials)(第 1 步), 但是他们会共用一套客户端证书。在 API 采用了这项技术以后, 客户端想要发起 HTTP 请求的话, 就必须同时提供自己的客户端证书以及一组用户证书。

注 1 如果你忽略了这个建议并在你的 API 中采用 session, session ID 就变成了一种在每个请求中都会被提交的临时证书。你这么做只是在现有证书系统之上又增加了一个很复杂的层次。

WWW-Authenticate报头和Authorization报头

我将介绍 3 种流行的认证技术。首先，我将讨论这 3 种技术的共同点：HTTP 的认证报头。

和第 1 章的故事相同，我们的故事也是从女主人公 Alice 开始的，我们的女主人公 Alice 为了获取某个表述而发起了一个简单的请求：

```
GET / HTTP/1.1
Host: api.example.com
```

但是这一次，服务器拒绝提供所请求的表述，而是提供了一个错误消息：

```
401 Unauthorized HTTP/1.1
WWW-Authenticate: Basic realm="My API"
```

401 响应码表示这是一个授权请求。WWW-Authenticate 报头说明了服务器所能接受的认证类型。在这个例子中，服务器想要客户端使用 HTTP Basic 认证。

Alice 需要通过某种方式获得某些证书。获取方式的细节内容依赖于所使用的认证机制。一旦她获得了她的证书，她就可以再次发起 HTTP 请求，将她的证书附加到 Authorization 请求报头中一并发送出去：

```
GET / HTTP/1.1
Host: api.example.com
Authorization: Basic YWxpY2U6cGFzc3dvcmQ=
```

这一次，服务器有希望为 Alice 提供她所请求的表述。

Basic认证

251

HTTP Basic 认证是由 RFC 2617 进行说明的。它是一种简单的用户名/密码方案。API 的用户应该提前设置一套用户名和密码——很可能是在其附属网站上注册一个账户，或者发送电子邮件来申请一个 API 账户。并没有相关标准来说明应该如何申请一个给定网站的用户名和密码。

不论发生什么情况，一旦 Alice 拥有了自己的用户名和密码，她就可以再次发起他原来的那个 HTTP 请求。这一次，她使用用户名和密码来为请求报头 Authorization 生成了一个值，就如同前面一节中看到的那样。

她通过了服务器的认证，服务器接受了请求并提供了一个表述以取代之前的 401 错误：

```
HTTP/1.1 200 OK
Content-Type: application/xhtml+xml
...
```

Basic 认证是很简单的，但是它有两个大问题。第一个就是它不够安全。YWxpY2U6cGFzc3dvcmQ= 看起来像是个加密的没有意义的数据，但是它实际上是对字符串 `alice:password` 进行了简单可逆的 Base64 编码转换^{注2}。这意味着任何监听 Alice 的网络连接的人都可以知道她的密码。他们可以发送包含 `Authorization:Basic YWxpY2U6cGFzc3dvcmQ=` 的请求来冒充 Alice。

如果 API 采用 HTTPS 而非普通的 HTTP，那么这个问题就不存在了。那些监听 Alice 的网络连接的人可以看到她打开了一个连接，但是请求和响应都通过 SSL 层进行了加密。

RFC 2617 还定义了另外一种称为 Digest 的认证方式，即便在没有采用 HTTPS 的情况下，Digest 方式也可以避免上述问题。我不会在本书中对 Digest 展开介绍，因为 Digest 和 Basic 认证方式都同时存在第二个问题。这个问题在万维网中并不是很大的事情，但是在 API 世界中却非常严重。这就是：使用 API 的人通常不会信任他们的客户端。

为了突出这个问题，设想一下非常流行的 API 比如 Twitter API。这套 API 非常流行，Alice 有 10 个各不相同的客户端使用到了这套 API。它们有的是运行在手机上的，有的是运行在桌面计算机上的，Alice 还允许了一些不同的网站来代表她使用这套 API（这种情况一直都在发生）。

这里有 10 种不同的客户端。如果其中的某个客户端变得行为异常并开始向 Alice 的账户发送大量垃圾信息该怎么办呢（这种情况也经常发生）？

252 伴随在这样的攻击后的是，Alice 必须修改她的密码。她只有这样才能让这个流氓客户端不再拥有合法的证书。但是她为这十个客户端都设定了相同的密码。另外 9 个客户端还是可靠的，但是对密码的修改会破坏所有的这 10 个客户端。在修改了密码之后，Alice 必须执行她那另外 9 个正常的客户端并将新的密码设置进去。如果这 9 个客户端中又有客户端变得行为异常，她就必须再次修改她的密码，并执行剩下的 8 个正常的客户端来再次一个个地将她新的密码设置进去。

如果 Alice 在一开始就能够为每个客户端设定一套不同的证书，就不会出现上面的问题了。这就需要 OAuth 出场了。

OAuth 1.0

在 OAuth 中，Alice 为每个客户端提供一套单独的证书。如果她决定不再使用某一个客户端，那么她会撤销这个客户端的证书，而另外 9 个客户端不受任何影响。如果客户端变得行为异常并以其用户的名义发送垃圾邮件，服务提供方可以进行干预并为这个客户

注2 Base64 是 RFC 2045 的 6.8 节定义的。大部分编程语言在它们的标准库中都拥有一种 Base64 实现。

端的每个实例（Alice 以及其他每个人的客户端）撤销证书。

OAuth 现在有两个版本。OAuth 1.0（由 RFC 5849 定义）能够有效地允许那些面向消费者的网站的开发人员集成你的 API。当你想要允许你的 API 能够集成到桌面应用、手机应用或者内嵌在浏览器的（in-browser）应用时，它就瓦解了。OAuth 2.0 和 1.0 很相似，但是它定义了处理上述场景的方法。

我将使用 OAuth 1.0 来简单介绍一些 OAuth 背后的概念，你可以阅读 Boyd 的《Getting Started with OAuth 2.0》（O'Reilly）来获取关于 OAuth2.0 的更多详细说明。

下面的 401 响应码看起来是表示服务器要求客户端提供一套 OAuth 证书：

```
HTTP/1.1 401 Unauthorized
WWW-Authentication: OAuth realm="My API"
```

获取这些证书是一个很复杂的流程。让我们假设 Alice 正在使用网站 *YouTypeItWePostIt.com*。她看到一个超媒体控件告诉她，她可以集成她在 *Example.net* 上的账号到她在 *YouTypeItWePostIt.com* 上的账号。她可以在不将 *Example.net* 上的密码告诉给 *YouTypeItWePostIt.com* 的情况下完成这件事情。



图 11-2 *YouTypeItWePostIt.com* 提示使用 *Example.net* 通行证登录

对 Alice 而言，这看起来是个很棒的主意，所以，她单击了那个按钮来激活那个超媒体控件。下一步会发生什么事情呢？

1. *YouTypeItWePostIt.com* 偷偷地从 API 提供方 *api.example.net* 申请了一组临时证书。这一步并不要求 Alice 的任何参与。
2. *YouTypeItWePostIt.com* 服务器向 Alice 的浏览器发送了一个 HTTP 重定向响应。Alice 离开了她正在访问的网站，最终停在了某个由 API 提供方 *Example.net* 所提

供的网页上。

如果 Alice 还没有在 *Example.net* 登录过，她需要登录或者创建一个新的账号。这意味着输入她的密码——但是需要注意的是，她是在将 *Example.net* 的密码提供给 *api.example.net*，而非 *YouTypeItWePostIt.com*。

3. 在登录进去以后，Alice 会看到一个网页，这个网页是和在第一步中所获取的那个临时证书绑定的。这个网页上的人类可读的文字向 Alice 说明将要发生什么事情，并询问她是否愿意将一组 *api.example.net* 的令牌证书授予 *YouTypeItWePostIt.com*（见图 11-3）。
4. Alice 做出决定之后，她的浏览器就重定向回她最初访问的网站：*YouTypeItWePostIt.com*。
5. a) 如果 Alice 在第四步中选择了“no”，客户端就不走运了。它不会从 Alice 那里得到任何 *api.example.net* 的令牌证书。
b) 如果 Alice 在第四步中选择了“yes”，客户端就被允许用它在第一步中获取的临时证书来换取一组真正的令牌证书。这些证书可以用于以加密的方式对 HTTP 请求进行签名，生成和下面请求类似的 Authorization 报头：

GET / HTTP/1.1

Host: api.example.net

Authorization: OAuth realm="Example API",
oauth_consumer_key="rQLd1PciL0sc3wZ",
oauth_signature_method="HMAC-SHA1",
oauth_timestamp="1363723000",
oauth_nonce="JFI8Bq",
oauth_signature="4HBjJvupgIYbeEy4kE0LS%Ydn6qyV%UY"

这时候，客户端就可以像是 Alice 本人一样正常使用 API 了。

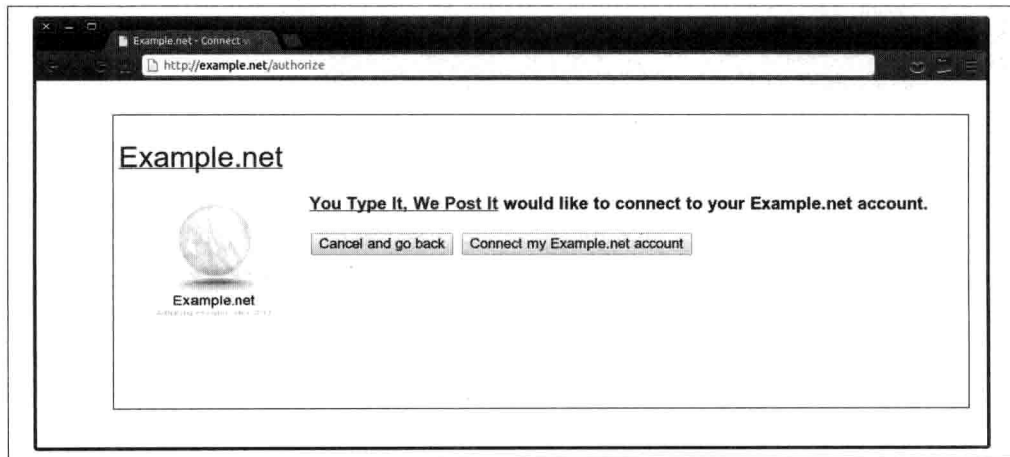


图 11-3 *Example.net* 为 *YouTypeItWePostIt.com* 请求 Alice 的证书

取决于 API，令牌证书可以是永久的，也可以在一定时间后自动过期（一旦证书过期，如果客户端想要继续使用这个 API，它就需要让 Alice 再次完成上述流程）。这个令牌证书可以允许客户端访问所有 Alice 使用 API 所能做的工作，或者它们也可以只让客户端获取部分权限（一种常见的限制就是对 API 的只读访问权限）。这些内容并没有在 OAuth 标准中进行声明，必须由 API 提供方来进行定义。

你可以看得到 OAuth 要比 HTTP Basic 认证复杂得多，但是——我需要特别强调——它避免了 Alice 必须将密码提供给她并不信任的 10 个不同的软件的情况的发生。OAuth 的复杂性还提供了一些其他有用的功能：

- 如果 Alice 不再希望某个客户端代表她做一些事情，她可以撤销它的令牌证书。
- 如果 API 提供方发现某款客户端软件行为异常，提供方可以撤销它的客户端证书（也就是 `oauth_consumer_key`）。这就意味着 API 停止为该款客户端的所有副本提供服务。
- 不同于 HTTP Basic 认证（但是和 HTTP Digest 类似），OAuth 1.0 可以在不暴露 Alice 的证书的情况下被应用到不安全的 HTTP 上。令牌证书是创建 Authorization 报头的 `oauth_signature` 部分所必需的数据，但是这个证书并不真正出现在这个报头中。服务器（知道 Alice 的证书的那个服务器）可以验证一个请求是否是由 Alice 的证书签名过的，但是监听这个请求的人并不能知道她的证书到底是什么。
- Authorization 报头中的 `oauth_timestamp` 和 `oauth_nonce` 的值阻止了“重放攻击 (replay attack)”，在这类攻击中，攻击方会监听 Alice 的请求，然后模拟它发起相同的请求（HTTP Digest 也可以阻止这类攻击）。

OAuth 1.0的缺点

当所有的活动都发生在 Alice 的 web 浏览器中时，OAuth 1.0 是非常棒的。但是如果 Alice 使用的是桌面应用时，该怎么办呢？

在这种情况下，Alice 需要临时切换到 web 浏览器上。在第二步中，不再重定向到 *api.example.net* 上的网页，取而代之是，桌面应用会打开一个显示这个网页的浏览器窗口。一旦 Alice 在第四步做出了自己的决定，也不存在 *api.example.net* 需要重定向回到某个地方。在后台，桌面应用需要继续询问 *api.example.net* 来确定 Alice 是否已经授权（或者拒绝）那个临时证书。

这就是在不获取用户名 / 密码作为输入来将 API 集成到桌面应用的问题 OAuth1.0 所给出的解答。在使用桌面应用时突然弹出一个 web 浏览器显得有点不合时宜，但这是可行的。不幸的是，在一些其他场景中，我前面讲过的五步流程是不够的，或者一点也不行：

- 如果 Alice 正在手机上使用 APP，或者在游戏机中玩一款游戏，会怎么样呢？突然在这些设备中弹出一个浏览器窗口会更加突兀。或许这甚至都不可能实现。有一些设备都并不具备 web 浏览器。
- 如果 Alice 是一款使用了 *api.example.net* 的软件客户端的作者，那又会怎么样呢？让她去获取一套临时证书并询问她是否她愿意授权给这个客户端，这是否有意义呢？
- 如果 Alice 正在使用的桌面应用恰好运行在她的 web 浏览器中，会怎么样呢？临时证书是否真的有必要呢？在第 5 步中，*api.example.net* 不可以仅仅提供一个包含真正的访问令牌的页面并让内嵌在浏览器的应用使用 JavaScript 代码来读取它吗？

256 ▶ OAuth 2.0 就是设计用来处理这些用例的。

OAuth 2.0

OAuth 2.0 是由 RFC 6749 定义的。它声明了 4 种获取 OAuth 访问证书的不同的流程（再次说明，我不会展开很详细的介绍，请参见《Getting Started with OAuth 2.0》一书来了解更多内容）：

- “认证码 (authorization code)”（见 RFC 6749 的 1.3.1 节）。这是我在 OAuth1.0 中所描述的方法。“资源所有者”（Alice）通过“认证服务器”进行授权（登录到 *Example.net*），然后“认证服务器”会重定向到提供访问令牌 (access token) 的“客户端”（*api.example.net*）。
- “隐式授权 (implicit grant)”（见 RFC 6749 的 1.3.2 节）。对于那些运行在 web 浏览器内部的应用是一个很好的选择。在 Alice 登录到 *Example.net* 以后，她会重定向到 *api.example.net*，进而又被重定向到一个包含访问令牌的 URL。这个过程不需要获取临时证书；内嵌在浏览器的应用可以通过浏览器的地址栏就读取访问令牌。
- “资源所有者密码证书 (resource owner password credentials)”（见 RFC 6749 的 1.3.3 节）。也就是，Alice 向客户端提供她在 *Example.net* 上的用户名和密码，客户端使用这些信息来换取一个 OAuth 访问令牌。
- 这恰好是 OAuth 所试图避免的：Alice 向自己不信任客户端提供自己的密码。但是在手机或者游戏机上，并没有其他更好的选择。
在这种情况下，恶意软件会窃取 Alice 的密码。但是合法的客户端会在获取到一个 OAuth 访问令牌以后就主动丢弃 Alice 的密码。这意味着合法的客户端不会在 Alice 因为其他原因修改了密码以后失效。
- “客户端证书”（RFC 6749 第 1.3.4 节）。当 Alice 是她自己的客户端的作者时，这可以避免很多麻烦。当 Alice 在 *api.example.net* 上注册了她的客户端后，她就自

动获得了一组证书，这些证书可以让她的客户端能访问她自己在 *example.net* 上的账号。

作为 API 提供方，你不需要实现上述全部 4 种应用流程。如果你编写的 API 是作为某款手机应用的后台而对外提供服务的，你可以仅仅实现“资源所有者密码证书”方案。但是如果你希望第三方能够将你的 API 集成到他们的客户端里，你就需要实现那些客户端希望使用的认证方案。

何时不采用 OAuth

考虑到 OAuth 标准的复杂性以及它们不同的应用方案，人们很容易放弃采用这一标准，转而采用 HTTP Basic 认证或者 HTTP Digest 认证来保护自己的 API。我建议你要坚持到底，并了解 OAuth 是如何工作的。如果你需要的话，可以学习和复制某个名气比较大的 OAuth 提供方，比如 Facebook 的实现方案。

◀ 257

OAuth 最基本的优势——将 Alice 在 *Example.net* 上的用户名和密码与 *api.example.net* 的证书相分离——真的是一个非常重要的作用。在我看来，只有在下面两种场景中你可以不采用这一认证方案：

- 你的 API 是毫无意义的供娱乐消遣用的。如果某个恶意软件窃取了 Alice 的证书，这也不会造成任何真正的破坏。
- 你的 API 的每个用户都将自己编写属于他自己的客户端。这样将 Alice 这个客户端开发人员与 Alice 这个最终用户进行分离就没有任何安全上的价值了。这意味着，当 Alice 在网站上修改她的密码时，她也需要修改她全部的 API 客户端上的密码。这应该不是一个大问题。

如果你理解了安全议题，除非你不认为你的 API 将会非常流行而使得恶意软件成为问题，你就应该在任何情况下都使用 OAuth。一旦你开始使用 HTTP Basic 认证，未来把所有的客户端都切换到 OAuth 将是非常困难的。不要让自己锁在成功的门外^{注 3}。

HTTP 扩展

相当多的 web API 按照其自己的定义都是基于 HTTP 协议的。但是 REST 的相关概念并不要求使用 HTTP，最多是超媒体约束要求你提供 HTML 表述。

下面会介绍两个 HTTP 扩展以及 3 个主要协议，这两个 HTTP 扩展分别描述了两个专门

注 3 当 Twitter API 在 2010 年从 Basic Auth 切换到 OAuth 时，开发人员将这一事件称为“OAuth 启示录”（OAuthpocalypse）。

供 API 使用的新方法,而 3 个协议则都以 HTTP 作为起点。这几种协议中,CoAP 并不常见,我将用一章的篇幅来进行介绍(第 13 章)。WebDAV 和 HTTP2.0 都紧紧基于 HTTP,所以我会在下面介绍一些扩展 HTTP 方法时一起进行介绍。

PATCH 方法

- 出处: RFC 5789 和其他
- 协议语义: 既不安全也不幂等

我在第 3 章中将这一方法作为供 API 开发人员使用的工具包的一部分进行过介绍。PATCH 方法可以解决 HTTP PUT 的性能问题。PUT 方法用一个新的完整的表述来代替资源的整个表述。这意味着即便只有一个很小的修改,客户端也必须重新发送整个表述。而 PATCH 方法则可以让客户端只发送它想要修改的那部分内容。

PATCH 方法的缺点就是客户端和服务端必须为补丁文档(patch document)商定一种新的媒体类型。幸运的是,你不需要自己提出这种格式。RFC 6902 为 JSON 定义了一种标准的补丁格式(patch format)。RFC 5261 则为 XML 文档定义了一种补丁格式,而互联网草案“draft-wilde-xml-patch”更为这种格式的文件注册了媒体类型 `application/xml-patch+xml`。

LINK 和 UNLINK 方法

- 出处: 互联网草案“snell-link-method”
- 协议语义: 幂等但不安全

LINK 方法用于创建两个资源之间的连接。当资源 A 链接到资源 B 时,一个链接到 B 的超媒体链接就会出现在 A 的表述中。

但是这个链接是如何创建的呢?一个 HTTP 请求如何才能引用两个不同的资源呢?当然是通过超媒体。向资源 A 的 URL 发送 LINK 和 UNLINK 请求时,资源 B 会出现在 Link 报头中。Link 报头上的链接关系则描述了所期望 A 和 B 之间的关系。

如下请求是将一个现有的条目添加到某个集合中(这是在集合模式中很常见的用例,但是 AtomPub 和 Collection+JSON 并没有对其进行定义):

```
LINK /collections/a6o HTTP/1.1
Host: www.example.com
Link: <http://www.example.com/items/4180>;rel="item"
```

如下请求是要求将某个资源链中的第二个资源移走:

```
UNLINK /story/part1 HTTP/1.1
Host: www.example.com
Link: <http://www.example.com/story/part2>;rel="next"
```

在处理完这条请求以后，在 */story/part1* 和 */story/part2* 上的资源还是存在的。只是它们之间的那个链接关系为“next”的链接不再存在了。可能，*/story/part1* 现在拥有一个链接到 */story/part3* 的 `rel="next"` 的链接。

从技术上来说，这些方法是不必要的。你可以使用 PUT 来完成它们相同的功能。但是，它们对事物进行了简化。它们抽象出一种常见的操作——控制资源之间的超媒体链接——并给它提供了协议语义。

在第 3 章中，我提到过在 1997 和 1999 年间，这些方法曾经是 HTTP 标准的一部分。RFC 2616 将它们从标准中删除了，原因就是在那个时候，人们不清楚为什么要使用它们以及应该如何使用它们。随着 web API 的发展以及 Link 报头的引入，人们现在清楚多了。

◀ 259

唯一阻止我建议大家使用 LINK 和 UNLINK 的因素就是，说明这些方法的互联网草案还没有被批准成为一个 RFC。

WebDAV

- 出处：RFC 4918 和其他
- 协议语义：文件系统操作

WebDAV 的目标是易于将远程文件系统上的文件和目录发布为 HTTP 资源。WebDAV 定义了许多新的 HTTP 方法和其他扩展，这使得它几乎被认为是一种不同的协议。

WebDAV 最引人注目的应用就是 Microsoft 的 Sharepoint 以及版本控制系统 Subversion。我们并不真的将那些应用看作是 API，大部分起到了远程文件系统作用的 API（Amazon 的 S3、Dropbox 的 API，等等）都并没有使用 WebDAV。它们都是使用 PUT 等标准 HTTP 方法的 fiat 标准，它们使用特定的 XML 或者 JSON 表述来提供元数据（metadata）。换句话说，它们才是和当今其他的 API 是一样的。

和 AtomPub 一样，WebDAV 也是一种被广泛忽视的开放标准，这是因为它和现代观念里人们所理解的 API 的样子并不一致。但是理解 WebDAV 还是很有用的，因为它是 API 领域的先驱者。下面是 WebDAV 的一些有趣的功能。

- WebDAV 通过定义一些“collection”资源实现了集合模式（第 6 章），这个资源类似于本地文件系统的目录。这些资源会响应 GET 和 DELETE 请求。WebDAV 同样定义了一些全新的 HTTP 方法：MKCOL，它是用于创建新的 collection 的方法。

- 客户端可以通过向某个 URL 发送一个 PUT 请求来上传文件，这个 URL 可以是客户端为新资源所选择的任何 URL。RFC 5995 是一个扩展，它允许客户端使用 POST-to-append 方法来上传新文件。在这种情况下，URL 是由服务器为新生成的资源选择的，而不是由客户端来选择。
- 你本地文件系统的文件除了包含数据外，还同样包含一些相关的元数据：文件名、文件创建日期，等等。WebDAV 资源会将这些元数据作为“属性”来表示，比如 `displayname` 和 `creationdate`。

WebDAV 定义了 HTTP 方法 `PROPPATCH` 用于修改资源的属性、`PROPFIND` 方法用于查找某个 collection 来展示具有某种属性的资源。

- WebDAV 允许客户端显式地对某个资源上锁（使用新的 HTTP 方法 `LOCK` 和 `UNLOCK`），这样其他客户端就不能访问该资源了。它可以和我在本章前面介绍的技术一起用于避免更新丢失问题。
- WebDAV 定义了新的 HTTP 方法 `MOVE` 和 `COPY`。它们的作用等价于文件系统上的操作。`MOVE` 用于更改某个资源的 URL，`COPY` 用于在其他的 URL 上保存一份资源的当前表述的副本。

WebDAV 同样为这些方法定义了新的超媒体控件：`Destination` 请求报头。这个报头包含了资源的新 URL，或者用于复制的 URL。相当于 `Link` 请求报头在 `LINK` 和 `UNLINK` 方法中的用途。

- WebDAV 定义了 5 个新的 HTTP 状态码。即便你不使用 WebDAV，它们中有一些也是很有吸引力的，比如 423 (`Locked`) 和 507 (`Insufficient Storage`)。但是我不建议在 WebDAV 之外使用 WebDAV 的功能。你应该求助于标准状态码。你可以使用 409 (`Conflict`) 来代替 423 (`Locked`)，并在问题细节文档中提供额外的上下文信息。

HTTP 2.0

- 出处：互联网草案“draft-ietf-httpbis-http2”
- 协议语义：和 HTTP 1.1 相同

HTTP 2.0 是当前版本的 HTTP 的继承者。当前版本的 HTTP 是由 RFC 2616 以及它的替代 RFC 定义的。HTTP 2.0 基于 SPDY 协议的，而 SPDY 协议是由 Google 定义的公司标准，它在 HTTP 上面增加了一个性能优化层 (performance layer)。大部分 web 浏览器现在已经支持 SPDY 协议，并且许多大型网站可以在客户端支持的情况下采用 SPDY 协议来提供数据。

HTTP 2.0 的目标是在保留协议语义的情况下，提高 HTTP 的性能。尽管它的名字叫 2.0，但是 HTTP 2.0 并不会引入一些会动摇 API 设计的根基的新功能。不管你是在开发 API、API 客户端，还是网站，你都应该能够伪装成在使用 HTTP1.1，并让兼容层能自动在 HTTP 1.1 和 2.0 之间进行转换。

当我在写这本书时，HTTP 2.0 还处于早期的开发阶段，准确说 HTTP 2.0 将如何工作还为时尚早。它可能最终和 SPDY 协议一点也不相似。但是它需要解决那些 SPDY 协议已经解决了的问题，这意味着，它很可能有下面这两个特性：

- HTTP 2.0 将通过压缩 HTTP 报头来节省带宽，这在 HTTP 1.0 中并不合法。
- HTTP 2.0 的客户端将能够通过一个 TCP 连接向服务器同时发送多个请求（“streams”）。这类似于 HTTP 1.1 的 pipelining 功能，但是正如我早前提到的那样，pipelining 并不会对性能提升有很大帮助。HTTP 2.0 需要包含一个和 pipelining 类似的真正有用的功能。

261

这些技术改进对 API 设计将有非常大和积极的作用。人们将不再需要那些将多个请求捆绑成一个请求来提高性能的常见的 API 设计模式^{注 4}。这些模式允许客户端使用单个 HTTP 请求来获取（更新）许多资源的表述。并不存在将这些“虚拟请求”打包的标准方式，但是这么做会节约非常多的时间，因为每个 HTTP 1.1 请求有很大的初始化设置的开销。

HTTP 2.0 会摆脱初始化配置的开销。发起 20 个 HTTP 请求，获取 20 个响应，将几乎和发起一个大的请求并获取一个大的响应一样快。我们将不再需要 API 来实现特定的批处理功能。

注 4 对于这些模式更好的介绍，请参阅《RESTful Web Services Cookbook》(O’ Reilly) 一书的第 11 章。

资源描述和Linked Data

就我在本书中所提到的那些数据格式而言，其主要用途便是让资源能够使用这些格式来谈论它们自己。这就是说，客户端在向资源的 URL 发送一个 GET 请求之后，便会接收到恰好是这个资源的一份表述。我把这种策略称为表述策略。

但是资源 A 的表述可能也会讨论到资源 B。下面这个简单的 Collection+JSON 文档便是来自一个资源（一个集合）的表述，但是它还讨论到了两个其他的资源（集合中的子项）：

```
{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/",

    "items" : [
      { "href" : "/api/messages/21818525390699506",
        "data": [
          { "name": "text", "value": "Test." }
        ]
      },

      { "href" : "/api/messages/3689331521745771",
        "data": [
          { "name": "text", "value": "Hello." }
        ]
      }
    ]
  }
}
```

我将这种策略称为描述策略。按照这种描述策略，一个表述会花费它大部分的时间来讨

论该表述所属资源以外的资源。

264 所有的超媒体格式在一定程度上都混合了表述和描述这两种策略，但是存在着一系列侧重专注于描述策略的格式：那些受到资源描述框架（Resource Description Framework，简称 RDF）数据模型的启发并且与语义网运动（Semantic Web movement）相关联的格式。

我并没有在第 10 章谈到这些格式，因为从 REST 的观点来看，它们是非常怪异的，一个纯粹的描述策略违反了 Fielding 约束。从 REST 的视角来看，RDF 文档通常描述的资源“并不存在”。要理解这些文档究竟在讲些什么，你必须采用不同的方式来思考。

幸运的是，第二波称为 Linked Data 的语义网运动旨在重新聚焦于表述策略上的 RDF。这是一个好消息，因为有多多个非常有用的数据格式是来自 RDF 的，并且还有一个非常强大的可用于创建机器可读的 profile 的语言：RDF Schema。

但是在我开始讨论 Linked Data 之前，你需要理解 RDF。这一章中的每一件事物都是基于 RDF，或者受启发于 RDF，又或者是针对 RDF 数据模型而设计的。你需要理解 RDF 文档是什么样的，它们的意思是什么以及 Linked Data 运动的必要性。

RDF

- 媒体类型：application/rdf+xml、text/turtle, 等等
- 定义方式：W3C 开放标准，定义见 <http://www.w3.org/standards/techs/rdf>，尤其是 <http://www.w3.org/TR/turtle/>
- 媒介：普通文本、XML、HTML 等
- 协议语义：通过 GET 进行导航
- 应用语义：无

这是 Maze+XML 迷宫中的单元格的一份 RDF 描述：

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:maze="http://alps.io/example/maze#">
  <rdf:Description about="http://example.com/cells/M">
    <maze:title>The Entrance Hallway</maze:title>
    <maze:east resource="http://example.com/cells/N">
    <maze:west resource="http://example.com/cells/L">
  </rdf:Description>
</rdf:RDF>
```

我们有着很多编写 RDF 的方式，包括一个叫作 RDFa 的 HTML 版本，以及一个叫作 Turtle 的普通文本版本。你看到的只是 XML 版本，叫作 RDF+XML。下面是迷宫单元格的一份 Turtle 描述，它所表达的意思与上一份表述相同：

```

<http://example.com/cells/M> <http://alps.io/example/maze#title> ↵
  "The Entrance Hallway" .
<http://example.com/cells/M> <http://alps.io/example/maze#east> ↵

<http://example.com/cells/N> .
<http://example.com/cells/M>↵<http://alps.io/example/maze#west> ↵

<http://example.com/cells/L> .

```

设置几个 Turtle 的简短形式，你将可以用一种更加紧凑的方式来表示相同的信息：

```

@prefix maze: <http://alps.io/example/maze#> .
<http://example.com/cells/M> maze:title "The Entrance Hallway" ;
                        maze:east <http://example.com/cells/N> ;
                        maze:west <http://example.com/cells/L> .

```

所有这 3 个文档描述的是相同的资源：一个具有 URI *http://example.com/cells/M*。在 RDF+XML 中，URI 在 `<Description>` 标签的属性中描述。在 Turtle 中，URI 位于一行开头的尖括号中（有些类似于 HTTP Link 报头将其目标 URL 放入了尖括号中这种方式）。

这些文档中每一个文档都对为它们所描述的资源提供了相同的声明（assertions）。每个文档都表明资源拥有一个叫作 *http://alps.io/example/maze#title* 的属性，该属性的值是字面量字符串 “The Entrance Hallway”。

每个文档同样也都说资源拥有两个属性，分别叫作 *http://alps.io/example/maze#east* 和 *http://alps.io/example/maze#west*。这些属性扮演了类似扩展的链接关系，它们的值是 URI（例如 *http://example.com/cells/L*）。它们说明了属性右侧资源与左侧资源之间的关系：

```

<http://example.com/cells/M> <http://alps.io/example/maze#west> ↵
<http://example.com/cells/L> .

```

如果将它翻译成人类的词汇的话，这一行 Turtle 是在说“单元格 L 在单元格 M 的西边”。

RDF 给了你一个用于讨论资源应用语义的框架。你可以讨论那些将一个资源连接到另外资源的链接关系，以及将自身与它的资源状态进行绑定的语义描述符。但是它不像其他的格式，RDF 属性不能是像“title”和“east”这样的简短字符串。它们只能是 URI，像 *http://alps.io/example/maze#title* 和 *http://alps.io/example/maze#east*。你可以通过使用 Turtle 前缀和 XML 命名空间来将 URI 缩短为 *maze:title* 和 *maze:east*，但是在幕后，它们仍然是 URI。

RDF将URL作为URI对待

所以，一个用于描述迷宫单元格的 RDF 文档会使用一系列例如 `http://alps.io/example/maze#title` 这样的属性。那么到底这些属性是什么意思呢？

266 在 REST 的世界里，这个问题有着一个非常明显的答案。一个 `http:` URL 标识了 Web 上的一个资源，如果你向该 URL 发送 GET 请求，你将会得到一个捕获了该资源状态的表述。

如果你向 `http://alps.io/example/maze` 发起一个 GET 请求，你将会得到一个 ALPS 文档。在文档中查找“east”和“title”，你将会得到它们的人类可读的说明信息。就说明信息本身还不足以消除语义鸿沟，但是如果你的自动化客户端因为无法理解某些内容而无法工作时，你作为开发者，便可以通过说明信息，从而知道去哪里修复问题。

但是 RDF 并不将 URL 作为 URL 来对待，它将它们作为 URI 对待，URI 一词自从第 4 章之后我便没有再提到过。一个 URI 标识了一个资源，就像一个 URL 一样，但是它并不保证你可以使用计算机来获取该资源的表述。这也是为什么我在本书中一直对 URI 的重要性轻描淡写的原因。正如我在第 4 章中所说的，使用 URI 来标识你的资源将无法满足多个 Fielding 约束。

但是我在我的 RDF 文档中使用了 `http:` URI。它们是 URL，所以我安全了，对吗？事实上，并不是，我并不安全。就 RDF 来说，就算 `http:` URL 也是 URI。URI `http://example.com/cells/N` 可能看上去很诱人，但是从 RDF 的角度来看，我们无法保证你向该 URI 发出 GET 请求后会返回给你一份表述。RDF 客户端可尝试发送请求并看看会发生什么，但是请不要假定任何事情一定会发生。

如果你向 `http://example.com/cells/M` 发送 HTTP GET 请求并从响应中获得了以上任何一种 RDF 文档，我们将会说你拥有了一份在 `http://example.com/cells/M` 的资源的“表述”。但是在现实生活中，如果你向该 URI 发送了一个 GET 请求，你将会获得一个 404 错误。`http://example.com/cells/M` 是虚幻的，是我为本书的讲解目的而构造的。所以我们不能说这些 RDF 文档是该资源的“表述”。从 REST 的角度来看，它们只是某个可能并不存在的资源的描述。

就 RDF 而言，这完全是合法的。为没有表述的资源编写描述当然是没有问题的。RDF 文档频繁提到的 `http:` URI 并不指向特定的任何事物。

在其他数据格式中，如果你看到一个链接，你便知道该文档在尝试告诉你关于可以发起的 HTTP 请求的相关信息。引用 Fielding 对超媒体的定义，链接是“应用控制信息”，它说明了你的 HTTP 客户端接下去可以做的事情。

但是就 RDF 规范而言，并没有被“控制”的“应用”。链接除了对抽象资源之间的抽象

连接进行命名之外，并没有提供别的作用。你无须访问这些链接来查看另一端的内容，便可以推断这些连接的相关信息。

什么时候使用描述策略

以 REST 的观点，这看上去相当疯狂，但是我们有几个不错的理由来支持我们使用资源描述策略。

首先，资源描述让你可以在无法控制表述的时候谈论资源。这很可能是因为资源由另一台服务器控制着（比如标识了你的某个用户的 OpenID 资源），或者是因为表述格式已经被固定。你可以使用描述策略来谈论其他人的资源。

其次，很多现存的 API 提供的表述并没有包含超媒体控件。向那些文档添加超媒体可能会破坏现存的客户端或违反某个标准。但是通过某个资源描述格式，你可以在某个不懂超媒体的文档顶部添加一套 phrase 的“外骨骼”注释：`[<phrase role="keep-together">hypermedia</phrase>]`。我在第 8 章曾提到过的 JSON-LD，也马上会在后续的章节中再次涉及，它就是设计用来满足这种目的的。

最后，你可以使用描述策略来讨论某个没有表述的资源，因为它并不存在于 Web 之上。我在第 4 章中曾提到的印刷版图书就是一个拥有众所周知的 URI：`urn:isbn:9781449358063` 的资源。你无法从该资源获取表述，但是你可以 GET 获取一个描述它的 RDF 文档。

让我们假设你向 `http://www.example.com/book-lookup/9781449358063` 发起了一次 GET 请求，并收到了以下的响应：

```
HTTP/1.1 200 OK
Content-Type: text/turtle

@prefix schema: <http://schema.org/> .
<urn:isbn:9781449358063> a schema:Book ;
                        schema:name "RESTful Web APIs" ;
                        schema:inLanguage "en" ;
                        schema:isbn "9781449358063" ;
                        schema:author _:mike ;
                        schema:author _:leonard .
_:mike a schema:Person ;
      schema:name "Mike Amundsen" .
_:leonard a schema:Person ;
      schema:name "Leonard Richardson" .
```

该实体消息体描述了某个资源：“《RESTful Web APIs》一书的印刷版”，由 URI

`urn:isbn:9781449358063` 标识。这个 URI 拥有一个很大的问题：你无法使用它来获取该资源的表述。这就是为什么当我们在设计 web API 的时候没有选择使用 URI 的原因。我们基于我们所控制的领域名称构造了 `http:` 或 `https:` URL。我们声明了那些与现实世界中的事物所对应的 URL，并且提供能捕获那些现实世界中事物状态的表述。

这并不是我在这里已经完成的。我还创建了第二个资源——“某个图书查找函数的输出信息”，由 URL `http://www.example.com/book-lookup/9781449358063` 标识。这个资源描述了由 `urn:isbn:9781449358063` 标识的现实世界中事物的资源状态，而不是尝试直接地表示资源。它的表述为现实世界中的事物提供了一系列的说明。它表明该现实世界的事物是一本书 (`aschema:Book`)，该书的标题是“RESTful Web APIs”，该书是采用英文编写的，并且具有两个作者，这些作者是人类 (`_mike a schema:Person`)，每个人都具有一个名字 (`schema:name`)……

这里有一个很棒的主意。如果有 10 个不同的组织定义了 10 个处理图书的 web API，我们将会最终为任何给定的 ISBN 生成 10 个不同的 URL。这将需要花费额外的工作来建立 `http://example.com/books/9781449358063` 的表述以及 `http://api.example.org/work?isbn=9781449358063` 这份讨论同一本书的表述。但是如果所有那些 URL 都提供描述 `urn:isbn:9781449358063` 的 RDF 文档，那么很明显就能发现所有的表述都在谈论同一个根本事物。

正如我所说，对于印刷体图书来说，这是一个很棒的主意。但是对于那些缺乏统一标识符的资源，该方法将不能很好地工作了。你可以使用 RDF 和 `schema.org` 词汇表来为人类提供所有种类的声明：

```
HTTP/1.1 200 OK
Content-Type: text/turtle

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix schema: <http://schema.org/> .
_:jennifer a schema:Person ;
    schema:name "Jennifer Gallegos" ;
    schema:birthdate "1987-08-25" .
```

该 RDF/Turtle 文档描述了某个资源。它说明了该资源是一个具有特定名字和生日的人。但是，我们没有一个约定的针对人的 URI 模式^{注1}，所以我们所讨论的资源并没有 URI。它只有一个内部标识“jennifer”。

这个文档是关于某个匿名资源的一系列声明。我们并没有约定俗成的方式来标识我们正

注1 `acct:` URI 模式，定义见互联网草案“draft-ietf-appsawg-acct-uri”，它不能用于标识人类，但是可以标识一个用户账户。对于很多 API 来说，这几乎已经足够了。

在谈论的人类。如果 10 个 API 都提供同一个人类的 RDF 描述，那么我们没有明显的方式来发现它们正在讨论的是同一个人。

此时此刻，你还是为你的 `person` 资源构造一个 `http:URL` 比较好。这样你便可以在某些人向该 URL 发起 GET 请求时提供一份该资源的表述。并且一旦你决定使用表述策略来替代描述策略时，你很可能会想使用一种比 RDF/XML 或 RDF/Turtle 具有更好的超媒体控件的数据格式。

资源类型

在语义网的各种最有用的思想中，有一条是这样的：资源可以被归类为一种或更多种的资源类型（同样也叫作抽象资源类型或语义类型）。这些与编程语言中的数据类型不同，它们只是一种分类（classification），例如区分图书的题材分类或动物的科属物种。

RDF 的 `type` 属性将一种类型分配给了一个资源。和 RDF 中的所有事物一样，资源类型由 URI 标识。下面是资源（`http://example.com/~omjennyg`）的一段描述，该资源被归类到了资源类型 `http://schema.org/Person`：

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
<http://example.com/~omjennyg> <rdf:type> <http://schema.org/Person> ;
                                <http://schema.org/birthDate> "1987-08-25" .
```

Turtle 定义了 `a` 作为 RDF 的 `type` 属性的简短形式，所以如果你只是谈论资源类型的话，无须为每个独立 Turtle 文件引入 RDF 词汇表：

```
<http://example.com/~omjennyg> a <http://schema.org/Person> ;
                                <http://schema.org/birthDate> "1987-08-25" .
```

有时“资源类型”的概念会从语义网络的世界渗透到 web API 这个更广阔的世界中。在 XLink 格式中（见第 10 章），链接可能会具有一个 `role` 属性，该属性会给该链接另一端的资源指定类型：

```
<a xlink:href="http://example.com/~omjennyg"
  xlink:arcrole="http://alps.io/iana/relations#author"
  xlink:role="http://schema.org/Person">
```

在 CoRE 链接格式（我将在下一章节中讨论）中，链接的 `rt` 属性也传达了另一端资源的类型（“`rt`”代表“resource type”）：

```
<http://example.com/~omjennyg>;rel="author";rt="http://schema.org/Person"
```


链接关系和资源类型是两种不同的事物。资源类型 (`http:// schema.org/Person`) 是关于链接另一端资源的陈述。而链接关系 (`author` 或 `http://alps.io/iana/relations#author`) 是关于链接本身的陈述——关于两个资源之间的关系的信息。

说到链接关系，有一个叫作 `type` 的并且已在 IANA 注册的链接关系，它允许表述对自身的资源类型做出声明：

```
HTTP/1.1 200 OK
Content-Type: text/plain
Link: <http://schema.org/birthDate>;rel="type"
```

1987-08-25

属性 `role` 和 `rt` 非常有用。它们可以让客户端提前查看链接另一端资源的类别，如果客户端它所看到的类别是它所期望的，它便可以继续前往访问该链接。

但是客户端应该如何知道它是否喜欢它所看到的类别呢？“资源类型”的概念来自 RDF，在 RDF 中，`http://schema.org/Person` 并不是一个 URL，它是一个 URI，一个没有意义的标示符。我们并不建议客户端向该 URI 发起 GET 请求。如果你想知道 `http://schema.org/Person` 和 `http://schema.org/birthDate` 是什么意思，你需要在某处找到用于描述这些资源的 RDF 文档。

如果没有了超媒体约束，我们就没有了用于查找描述文档的规则。你可能会不断地瞎逛，直到你找到它为止。但是该文档真的非常重要！这是用于消除语义鸿沟的唯一方式。如果没有了该文档，`http://schema.org/birthDate` 将只是一个简短而没有意义的字符串。

RDF Schema

目前，我们先暂时搁置如何找到该神奇文档的问题。一旦你找到该文档，它又是什么样子的呢？该文档将会对这些简短而无意义的字符串进行说明。它会表明 URI `http://schema.org/Person` 和 `http:// schema.org/birthDate` 基本上符合我们对“人”和“生日”的日常观念的认识。它扮演的角色类似于 `profile`，从原理上说，它跟 ALPS 或 XMDP 没什么区别。

RDF profile 有时被叫作词汇表 (vocabulary) 或本体 (ontology)。它是使用一种叫作 RDF Schema 的元词汇表写成的。RDF Schema 让你可以使用 RDF 来对资源类型做出声明，而不仅仅是单独的资源。

下面是一个说明 *http://schema.org/Person* 是什么的 RDF Schema 文档：^{注2}

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
<http://schema.org/Person> a rdfs:Class ;
    rdfs:label "Person" ;
    rdfs:comment "A person (alive, dead, undead, or fictional).";
    rdfs:subClassOf <http://schema.org/Thing> .
```

第三行表明了 *http://schema.org/Person* 的资源类型是 *rdfs:Class*。这意味着 *http://schema.org/Person* 并不代表某个特定的现实世界中的事物（一个单独的人），它代表的是一个概念（“人”的概念）。 271

属性 *rdfs:label* 和 *rdfs:comment* 指向了一些关于这些神秘概念的人类可读的信息。某个阅读了该文档的人将会获悉资源 *http://schema.org/Person* 对应了我们日常生活中“人”的观念。所以如果某些其他资源的类型是 *http://schema.org/Person*，那么说明该资源也对应了一个独立的人。

但是该 RDF Schema profile 同样还有一个机器可读的元素：属性 *rdfs:subClassOf*。这表明每个是 *http://schema.org/Person* 的资源同样也是一个 *http://schema.org/Thing*。你对 *http://schema.org/Thing* 究竟是什么好奇吗？你可以通过查看该 URI 的 RDF 描述来满足你的好奇心，该描述正好位于 *http://schema.org/Person* 描述所在的同一个文件中。

```
...
<http://schema.org/Thing> a rdfs:Class
    rdfs:label "Thing"
    rdfs:comment "The most generic type of item."
```

它表明 *http://schema.org/Thing* 没什么特别的。是不是很扫兴？但是至少现在你知道该 *http://schema.org/Thing* 与 1982 年 John Carpenter 的电影《The Thing》（中文译名：《怪形》），或者是 2011 年再拍的《The Thing》（中文译名：《怪形前传》）等电影都无关，与奇迹漫画超级英雄中名叫 Thing 的角色也无关。

那么 *http://schema.org/birthDate* 又如何呢？它又是什么意思呢？好的，下面就是该资源的一段 RDF 描述，是从作为我刚刚向你展示过的其他两段描述的同义词汇表文档中改编而来的：

```
<http://schema.org/birthDate> a <rdf:Property> ;
    rdfs:label "birthDate" ;
```

注2 schema.org 的官方 RDF 本体论位于 *http://schema.org/docs/schema_org_rdfa.html*。我希望可以在我的例子里直接引用该文档，但是该文档使用了 RDFa，这是 RDF 的 HTML 版本，我在此处没有太多空间来说明 RDFa。所以我对该文档做了一些转化，将它转成了我在本章中一直使用的基于文本的 Turtle 格式。

```
rdfs:comment "Date of birth." ;
rdfs:domain <http://schema.org/Person> ;
rdfs:range <http://schema.org/Date> .
```

该描述中的人类可读的部分只不过是确认了你刚才所怀疑的内容：<http://schema.org/birthDate> 是与现实世界中“生日”这一概念所对应的。此处真正有意思的是该描述中的机器可读的部分——**domain** 和 **range**：

```
rdfs:domain <http://schema.org/Person> ;
rdfs:range <http://schema.org/Date> .
```

RDF/Turtle 的这两行表明了 **birthDate** 是介于 **Person** 和 **Date** 之间的一种关系。可以将 **birthDate** 想象成一个函数，你传入一个 **person**，你将会得到一个 **date**——该 **person** 的生日。其中 **domain** 是输入项，而 **range** 是输出项。

272 这两项机器可读的声明是抛向语义鸿沟另一端的绳索。差不多 schema.org 的 RDF 词汇表中的每一项对人类来说都是显而易见的。但是想要通过编程来让计算机理解我向你所展示的词汇表的某些部分，就需要你教会计算机 4 种概念：**Person**、**Thing**、**birthDate** 和 **Date**。schema.org 词汇表包含了差不多有上千个概念，这样下去将会有更多的被添加进去，没有人能编程让计算机来理解所有这些概念。

这就是为什么 RDF Schema 词汇表中机器可读的部分（比如 **subclassOf**、**domain** 和 **range** 等属性）如此重要的原因。它们就应用语义方面给了计算机一种低层次的理解，而无须人类的辅助。一个并不知道 **birthDate** 是什么的 RDF 客户端可以凭借与人相关的某种日期的知识（从 **domain** 和 **range** 中获取的）来将某些功能应付过去。

ALPS 和 XMDP profile 重度依赖于应用语义中的人类可读的描述信息。这意味着它们依赖于人类劳动来将这些描述信息转化成可以工作的客户端代码。RDF Schema profile 将 API 中更多的应用语义转化成机器可读的方式。RDF Schema 的一项叫作 OWL（我将不会展开讨论）的扩展让你可以将这种想法实行得更加深入。我们的梦想是教会计算机数百个基本概念，然后让它自己识别出剩余的概念，从而取代教会计算机数百万特定概念的方式。

这样做的成本是这种说明方式会变得非常复杂。你可以使用 RDF Schema 和 OWL 并按照基本的概念来描述“生日”。这很可能会出现类似“某个事件的日期，该事件中某个人的状态从未存在变成了已存在”^{注3} 这样的描述。对于大部分应用来说，客户端作者编写一些代码来按照他们想要的方式处理生日会更加容易。

注3 BIO 词汇表并没有这么深入的描述，但是它确实对“生日事件”这一资源进行了描述，见 <http://purl.org/vocab/bio/0.1/Birth>，它将出生的人、他们的父母、日期以及地点等信息都汇集在了一起。

Linked Data运动

RDF 在使用机器可读的术语来表达应用语义方面是很棒的。但是你不能单独在 RDF 之上构造 RESTful API，因为 RDF 使用了 URI 来代替 URL。这样我们就无法保证客户端可以从它所看到的那些被描述的任何资源中获取表述。

当然，作为一名 API 设计者，你可以忽略掉那些规则。你可以声明所有你的 URI 都是 URL，并且你所有的资源都具有表述。你的资源可以提供专注于描述它们自身的 RDF 文档，而不是描述其他那些可能有或可能没有表述的资源。这样一来，你又重新获得了 Fielding 约束。在你的 RDF 表述中的 URI 都将成为超媒体链接，而你的客户端将在访问它们时感觉良好。

这一学说被称为 Linked Data。该术语出自 Tim Berners-Lee 在 2006 年的一篇论文，在这篇文章中他确定了将机器可读数据放置到 Web 上的 4 个原则。以 REST 的观点来看，这些原则放宽了 RDF 的 URI 约束（它们表明将 URI 作为 URL 处理也是没问题的），从而充分利用了 Fielding 约束。它们使语义网的哲学向 REST 哲学又靠近了一步。下面是 Berners-Lee 关于 Linked Data 的 4 个原则：

◀ 273

1. 使用 URI 作为事物的名字。

从 REST 的角度看，这说明一个 URI 标识了一个资源。在第 1 章中，我将这一点称为可寻址原则。

2. 使用 HTTP URI，这样一来人们便可以查找到那些名字。

这一点具有两个部分。首先，你不应该将你的资源标识为像 `urn:isbn: 9781449358063` 这样的 URI。你应该使用像 `http://example.com/books/9781449358063` 这样的 URL。的确，`urn:isbn:9781449358063` 是一种更为通用的引用资源的方式，但是因为它过于通用，以至于客户端无法使用该引用来做任意的事情。

其次，资源应该具备表述。向某 URL 发送了 GET 请求的客户端可能会获取到一些有用的数据。像 `http://vocab.org/vnd/ mamund.com/2013/numbers/primes` 这样的 URL 看上去很不错，重要的是当你向它发送一个 GET 请求后会得到一个 404 错误。然后你就会发现该 URL 实际上是一个 URI，它并没有表述。在某处可能有着一个描述该 URI 的神奇的文档，但是还是祝福你能幸运地找到它吧。

3. 当某些人在查找一个 URI 的时候，使用标准（RDF*，SPARQL）来提供一些有用的信息。

再一次的，资源都具有表述。向资源的 URL 发送了 GET 请求的客户端可能会接收到一

个捕获了该资源当前状态的文档。

确切的标准并不重要(我甚至不会在本书中讨论 SPARQL)。重要的是你使用了某些标准,而不是自行去定义一些自定义的数据格式。按照这种方式,理解你标准的客户端将会自动了解到如何去处理你所提供的数据——至少在一个基本的水平。这也是我在本书中一直强调的一个主题:你应该使用一个现有的超媒体格式,而不是去自行定义。

4. 包含链向其他 URI 的链接,这样客户端就能发现更多的东西。

最后,最大的回报就是:超媒体约束。URI 现在已经是 URL 了,它是一个链接,客户端可以通过访问它从而获得一份表述。该表述将会包含其他的链接,客户端可以通过访问它们从而接近任何它编程所需的内容。

274 ➤ 如果你想要编写一个 Linked Data API,我建议你使用 JSON-LD 来取代 RDF/XML 或 RDF/Turtle 作为你的表述格式。JSON-LD 是 RDF 的新的延续,特别设计用来构建类似今天的其他超媒体 API 的 Linked Data API。

JSON-LD

- 媒体类型: `application/ld+json`
- 描述方式: 进展中的开放标准,定义见 <http://www.w3.org/TR/json-ld/>
- 媒介: JSON
- 协议语义: 通过 GET 链接进行导航
- 应用语义: 非常灵活,但是每个文档都必须自己定义自己的应用语义

在第 8 章中,我曾将 JSON-LD 作为一种 profile 格式谈到过。我展示了一个单调的 JSON 表述……

```
HTTP/1.1 200 OK
Content-Type: application/json

{ "n": "Jenny Gallegos",
  "photo_link": "http://api.example.com/img/omjennyg" }
```

...could be transformed into a hypermedia document by the addition of a JSON-LD “context”:

……是如何通过添加 JSON-LD “上下文”从而被转化成一个超媒体文档的:

```
HTTP/1.1 200 OK
Content-Type: application/ld+json

{
  "@context":
```

```

{
  "n": "http://alps.io/schema.org/Person#name",
  "photo_link":
  {
    "@id": "http://alps.io/schema.org/Person#image",
    "@type": "@id"
  }
}

```

该 JSON-LD 上下文通过链接到说明信息，从而对链接关系 `photo_link` 和语义描述符 `n` 进行了说明。我展示过该上下文的不同版本：一个链接到了一个 ALPS profile，而另一个链接到了一个人类可读的文档，还有一个使用了一个由 RDF 词汇表描述的 URI。下面是另一个使用了 `schema.org` 词汇表的例子：

```

{
  "@context":
  {
    "n": "http://schema.org/name",
    "photo_link":
    {
      "@id": "http://schema.org/image",
      "@type": "@id"
    }
  }
}

```

275

将JSON-LD作为一种表述格式

到目前为止，我一直将 JSON-LD 作为一种 profile 格式来介绍：它是针对普通 JSON 文档的一种附加组件，用于对应用语义进行说明。你可以使用 Link 报头来将 JSON 文档和它的 JSON-LD 上下文连接起来：

```

Link: <http://api.example.com/profile.person.jsonld>;rel="http://www.w3.org/ns/json-ld#context"

```

说 JSON-LD 是一种 profile 格式并不是非常准确的。我们知道一个具有 JSON-LD 上下文的 JSON 文档并不只是给客户端带来一些关于它应用语义的附加信息。它完全改变了客户端应该如何处理文档的方式。如果脱离了上下文来解析一个 JSON 文档，那么你可以使用 JSON 的规则并最终会得到一个嵌套的数据结构。如果结合了它的上下文来解析同一个文档，你可以使用 RDF 规则，并最终会得到一系列 RDF 断言（RDF assertions）。

JSON-LD 并不只限于附加组件的角色。一旦你向任何 JSON 对象添加 `@context` 属性并将它以 `application/ld+json` 的媒体类型进行提供，它都会成为一个 JSON-LD 文档。

这意味着你可以将 JSON-LD 上下文与你所提供的数据进行绑定,并一次性提供全部的内容:

```
HTTP/1.1 200 OK
Content-Type: application/ld+json

{
  "n": "Jenny Gallegos",
  "picture_link": "http://www.example.com/img/omjennyg",
  "@type": "http://schema.org/Person",
  "@context": {
    {
      "n": "http://schema.org/name",
      "photo_link": {
        {
          "@id": "http://schema.org/image",
          "@type": "@id"
        }
      }
    }
  }
}
```

276

此时此刻, JSON-LD 成为了一种传统的超媒体格式。而 Link 报头也不再是必需的了, 因为这里仅仅只剩下了一个文档。始终清晰不变的是 photo_link, 它是一个超媒体链接。事实上, 这显得比之前更加清晰, 因为所有的信息都在一个地方了。

但是作为表述格式, JSON-LD 的能力还不是非常强大。多亏了它对 RDF 的继承, 从而使 JSON-LD 可以对应用语义进行非常详细的描述, 但是它的协议语义却仍然非常有限。JSON-LD 除了逐位地访问数据中的链接之外, 什么也做不了。客户端无法改变数据, 因为 JSON-LD 没有超媒体控件来触发非安全的 HTTP 请求。

如果你想在你的 API 中使用 JSON-LD, 我建议你同样也使用一种叫作 Hydra 的扩展。

Hydra

- **媒体类型**: application/ld+json
- **描述方式**: 进展中的个人标准, 见 <http://www.markus-lanthaler.com/hydra/>
- **媒介**: JSON
- **协议语义**: 完全通用
- **应用语义**: 来自 JSON-LD; 实现了集合模式 (“collection” 和 “resource”), 但是集合没有特别的协议语义

Hydra 是一个向 JSON-LD 添加了很多协议语义的 JSON-LD 上下文。通过它自身, JSON-LD 只允许你指定被 GET 请求触发的链接 (使用 “@type”: “@id”)。通过将 Hydra 添加到混合的格式中, 你可以指定几乎任意的 HTTP 请求。

下面是一个 Hydra 文档，它描述了跟 *YouTypeItWePostIt.com* 的博客 API 一样的应用语义和协议语义（让我们假设该文档由 <http://example.com/youtypeit.jsonld> 提供）。

```
{
  "@context": "http://purl.org/hydra/core/context.jsonld",
  "@type": "ApiDocumentation",
  "title": "Microblogging API",
  "description": "You type it, we post it.",
  "entrypoint": "http://example.com/api/",
  "supportedClasses": [
    {
      "@id": "#BlogDirectory",
      "title": "A directory of blogs",
      "description": "Links to all blogs.",
      "supportedProperties": [
        {
          "@id": "#blogs",
          "@type": "link",
          "title": "Blogs",
          "description": "The available blogs.",
          "domain": "#BlogDirectory",
          "range": "#Blog"
        }
      ]
    }
  ],

  {
    "@id": "#Blog",
    "@type": "Class",
    "subClassOf": "Collection",
    "title": "Blog",
    "description": "A collection of posts.",
    "supportedOperations": [
      {
        "@type": "CreateResourceOperation",
        "method": "POST",
        "expects": "#BlogPost"
      }
    ]
  },

  {
    "@id": "#BlogPost",
    "@type": "Class",
    "title": "Post",
    "description": "A single blog post.",
    "supportedProperties": [
      {
        "@id": "#content",
        "@type": "rdfs:Property",
```



```

        "title": "Content",
        "description": "The content of a blog post.",
        "domain": "#BlogPost",
        "range": "xsd:string"
    }
  ]
}
]
}

```

一个理解 JSON-LD 和 RDF Schema 的客户端可以从该文档中得到很多信息。它可以学习到 3 种资源类型 (<http://example.com/youtypeit.jsonld#BlogDirectory>、<http://example.com/youtypeit.jsonld#Blog> 和 <http://example.com/youtypeit.jsonld#BlogPost>)，每一项都拥有一份人类可读的描述信息。

278 即使客户端对 Hydra 毫不了解，也足以理解这段表述。下面是一份某 API 主页的 JSON-LD 表述，由 <http://example.com/api/> 提供：

```

HTTP/1.1 200 OK
Content-Type: application/ld+json

{
  "@context": {
    "blogs": "http://example.com/youtypeit.jsonld#blogs",
    "Blog": "http://example.com/youtypeit.jsonld#Blog"
  },
  "@id": "http://example.com/api/",
  "blogs": [
    { "@id": "/api/blogs/1", "@type": "Blog" },
    { "@id": "/api/blogs/2", "@type": "Blog" }
  ]
}

```

也许你想知道属性 `blogs` 的意义是什么？好的，`@context` 表明了它的应用语义定义在 <http://example.com/youtypeit.jsonld#blogs>，下面的内容便是：

```

{
  "@id": "#blogs",
  "@type": "link",
  "title": "Blogs",
  "description": "The available blogs.",
  "domain": "#BlogDirectory",
  "range": "#Blog"
}

```

它是一个博客的 list。更正式地说，它是一个函数，它的输出（`range`）都具有资源类型 <http://example.com/youtypeit.jsonld#Blog>。这意味着下面的 JSON 是对两个不同的 `blog-type` 资源的描述：

```

"blogs": [
  { "@id": "/api/blogs/1", "@type": "Blog" },
  { "@id": "/api/blogs/2", "@type": "Blog" }
]

```

当然，这并不算是一段描述。这些资源对你而言，除了它们的 URI 和语义类型之外，你别的什么都不知道。在一个传统的 RDF 文档中，故事可能就到此结束了。你将永远不会对这些资源有更多的了解，除非你在某处找到了一份关于它们的更好的描述。

但是这是一个 JSON-LD 文档，所以你知道它是遵守超媒体约束的。这也是在鼓励你向 `/api/blogs/1` 或 `/api/blogs/2` 发起 GET 请求。你知道如果你发起了该 GET 请求，将会得到一个满足 Blog 应用语义的表述。

到了这一步，已经完全不需要 Hydra 的相关知识了。JSON-LD 客户端可以向 API 主页发起 GET 请求，从而理解上下文，并向 `/api/blogs/2` 发起第二个 GET 请求。它可以将获取的表述与该博客资源类型的描述进行比对，从而理解该特定“Blog”的应用语义。

◀ 279

但是了解 Hydra 的客户端在此时却拥有着很大的优势，它理解一个叫作 `supportedOperations` 的特别属性。在这个上下文中，`supportedOperations` 表明了一个 Blog 类型的资源能像支持 GET 一样支持 HTTP POST。再看看这一段：

```

{
  "@id": "#Blog",
  ...
  "supportedOperations": [
    {
      "@type": "CreateResourceOperation",
      "method": "POST",
      "expects": "#BlogPost"
    }
  ]
}

```

这表明客户端可以通过向类型为 Blog 的资源发起 HTTP POST 请求从而创建一个新的资源（类型是 BlogPost）。请求的实体消息体应该是一份满足 BlogPost 应用语义的 JSON-LD 表述。

那么 BlogPost 的应用语义是什么呢？好的，原始的上下文表明了一个 BlogPost 拥有一个单独的属性，称为 `content`，它的类型是字符串（`xsd:string`）：

```

{
  "@id": "#BlogPost",
  ...
  "supportedProperties": [
    {

```

```

    "@id": "#content",
    "@type": "rdfs:Property",
    "title": "Content",
    "description": "The content of a blog post.",
    "domain": "#BlogPost",
    "range": "xsd:string"
  }
]
}

```

将这些信息汇总起来，Hydra 客户端便知道它可以发送一个如下的 POST 请求：

```

POST /api/blogs/2 HTTP/1.1
Host: www.example.com
Content-Type: application/ld+json
{
  "@context": {
    "content": http://www.example.com/youtypeit.jsonld#content"
  },
  "content": "This is my first post."
}

```

JSON-LD 为 API 提供者提供了一种方式来对那些看似普通的 JSON 文档的应用语义进行说明。一个 JSON-LD 上下文同样也可以对文档的协议语义进行说明，通过给予客户端不受限制的许可，从而让客户端可以向任何它们所发现的 URI 发起 HTTP GET 请求。但是就其本身而言，JSON-LD 只能描述安全的状态转换。

Hydra 则能更进一步。包含了特殊 Hydra 属性的 JSON-LD 上下文可以告诉客户端并允许它发起任意类型的 HTTP 请求，而不仅仅是 GET。Hydra 可以详细地描述非安全的状态转换。

总的来说，我将 Hydra 上下文与 WADL 文档以及 OData 元数据文档做了对比，我曾在第 10 章中提到过它们。这些文档往往趋向于预先定义好资源的类型（Blog、BlogPost），而不是在运行时表现出每个资源的行为。这原本也没有什么不妥。几乎任何的 API 都具有不同的资源类型，而特定类型的所有资源都将具有相似的应用和协议语义。

但是有着很大的诱惑使得我们对“资源的抽象语义类型”和“我的数据模型中的类的实现细节”发生混淆。Hydra 上下文、OData 元数据文档和 WADL 文档引诱着服务端 API 开发者基于他们的内部数据模型自动生成一次性的词汇表，而不是复用标准词汇表。

下面还有一个更大的问题，我曾在第 9 章提到过。因为这些文档并不会频繁变更，客户端 API 开发者会试图将它们作为服务描述文档来处理，从而能提供 API 应用语义的完整概览。用户将会尝试基于 Hydra 上下文产生客户端代码——但是客户端代码会在上下文

改变时被破坏掉。

在我编写此书时，Hydra 标准仍尚在制定中，但是对于基于 JSON 的超媒体 API 来说，Hydra 是相对于普通 JSON-LD 更好的选择，因为它可以描述非安全的状态转换。只要确保你所使用的方式不会破坏 REST 超媒体约束所带来的好处。

XRD家族

我之所以推荐 JSON-LD 是因为它同时采用了 Linked Data 和 REST 的基本原则。URI 应该是 URL，以及它们背后都应该具备有用的表述。像 *urn:isbn:9781449358063* 这样的 URI 所带来的麻烦远远超过了它们本身的价值。

如果你拒绝做出妥协会怎么样？伴随着一个使用纯描述策略的 API 你又能够走多远？这就是我想要在对 XRD 格式以及两个构建于其上的标准的讨论中所要进行的探索：Web 主机元数据文档（web host metadata documents）和 WebFinger。

◀ 281

对于当今 API 中所使用的特定 XML 和 JSON 格式来说，XRD 是它们采用策略模式时的答案。Web 主机元数据文档使得为围绕那些你无法控制的资源构建超媒体 API 成为可能，而且这些资源很可能是完全没有表述的。这可能看起来像一个毫无意义的技巧，但是 WebFinger 向我们展示了一个真实的使用案例。

XRD和JRD

- 媒体类型：application/xrd+xml 或 application/jrd+json
- 定义方式：RFC 6415 (JRD), 开放标准 (XRD) (见 <http://docs.oasis-open.org/xri/xrd/v1.0/xrd-1.0.html>)
- 媒介：XML 或 JSON
- 协议语义：使用 GET 链接进行导航
- 应用语义：无

XRD 是一种传统的基于 XML 的文档格式，它被设计用于描述来自外部的资源。它跟 RDF 不同，XRD 是通过语义描述符和链接关系进行区分的。其中语义描述符位于 `<Property>` 标签中，而链接关系位于 `<Link>` 标签中。

一旦你理解了描述策略就会觉得它很简单，下面是 Maze+XML 迷宫中的一个单元格的 XRD 描述：

```
<XRD xmlns="http://docs.oasis-open.org/ns/xri/xrd-1.0">
  <Subject>http://example.com/cells/M</Subject>
```

```

<Property type="http://alps.io/example/maze#title">
  The Entrance Hallway
</Property>
<Link rel="http://alps.io/example/maze#east"
      href="http://example.com/cells/N" />
<Link rel="http://alps.io/example/maze#west"
      href="http://example.com/cells/L" />
</XRD>

```

再次说明，由 `http://example.com/cells/M` 标识的资源是否存在并不重要（就是指是否具有表述）。这只是一个有着一些关于该资源信息的文档。

RFC 6415 定义了一种简单的方式来将 XRD 文档翻译成 JSON 对象。翻译后的结果我们称为 JRD，它是以 `application/jrd+json` 媒体类型提供的：

282

```

{
  "subject": "http://example.com/cells/M",
  "properties": {
    "http://alps.io/example/maze#title": "The Entrance Hallway"
  },
  "links": [
    { "rel": "http://alps.io/example/maze#east",
      "href": "http://example.com/cells/N" },
    { "rel": "http://alps.io/example/maze#west",
      "href": "http://example.com/cells/L" }
  ]
}

```

Web主机元数据文档

- 媒体类型：application/xrd+xml 或 application/jrd+json
- 定义方式：RFC 6415
- 媒介：XML 或 JSON
- 协议语义：通过 GET 进行导航；通过 GET 具有有限的查找能力
- 应用语义：无

Web 主机元数据文档是一个 XRD 文档，该文档将 API 的顶级描述作为一个整体包含进来。这就跟 JSON 主文档中的描述一样（见第 10 章）。Web 主机元数据文档的 XRD 版本假定位于 Well-Known URI `/.well-known/host-meta` 之下（见第 9 章对 Well-Known URI 的介绍）。如果有一个该文档的 JRD 版本，它会被假定位于 Well-Known URI `/.well-known/host-meta.json` 之下。

就像任意的 XRD 文档一样，一个 Web 主机元数据文档可以包含属性和链接。属性是将 API 作为整体后的属性，例如服务器实现的当前版本。而链接是链向 API 中特别重要的

资源的链接，例如顶级集合。

XRD 链接可以使用 `template` 属性来取代 `href` 属性。这样一来便将链接转变成了一项 URI 的目录查找服务。下面是一个展示了查找服务在 API 的高级描述中如何带来作用的简单例子：

```
<Link rel="copyright" template="http://example.com/copyright?resource={uri}" />
```

`<Link>` 标签表明，如果客户端想要查找任何资源的版本声明，并且不管该资源是由 `http:` URI 还是 `urn:isbn:` URI 标识的，它都应该将 URI 传递给具有 `rel="copyright"` 属性的模板并向其发送 GET 请求。GET 请求将会根据传入的 URI 触发 `copyright` 的状态转换。按照这种方式，客户端便可以触发没有表述的 URI 的状态转换。URI `urn:isbn:9781449358063` 没有表述，但是如果你向 `http://example.com/copyright?resource=urn:isbn:9781449358063` 发送 GET 请求的话，你便可以得到一个相关资源的表述：该书的版权信息。

◀ 283

属性 `template` 的值与 URI 模板非常相似，但它并不是 URI 模板，因为你可以使用的唯一的变量只有 `{uri}`。一个 Web 主机元数据模板可以使用 `<Link>` 来链接到一个特定的其他资源（使用 `href` 属性），它还可以链接到一个基于 URI 的查找服务（使用带有 `{uri}` 变量的 `template` 属性），但也就是这样而已。你无法在 Web 主机元数据文档中放入通用的搜索表单。

下面的例子在 RFC 6415 的作者们头脑中曾经清晰可见：

```
<Link rel="lrdd" href="http://example.com/lookup?resource={uri}" />
```

该标签告诉客户端，如果它想要获取资源的 XRD 描述，可以将 URI 填入模板，然后向生成的结果 URL 发起一个 GET 请求。此处的 `lrdd` 链接关系是一个在 IANA 注册的链向 XRD 描述的链接（它之所以叫 `lrdd` 的原因由于过于复杂和无趣，我们不再深入）。

Web 主机元数据文档现在是一份 XRD 文档，它告诉了你如何查找其他的 XRD 文档。你可能永远不能从某个 URI 获取表述，但是 Web 主机元数据文档可以帮助你找到关于该 URI 的各种各种的描述。

WebFinger

- 媒体类型：`application/jrd+json`
- 定义方式：互联网草案“draft-ietf-appsawg-webfinger”
- 媒介：JRD

- 协议语义：与 JRD 相同
- 应用语义：用户账户

WebFinger 协议只是一个供 JRD 文档查找关于用户账户信息用途的名字。一个账户可以采用 `acct: URI` 模式的方式^{注4}来使用邮箱地址进行标识：

```
acct:jenny@example.com
```

也可以由 `http: URL` 的方式来标识账户，这可能是一个由 OpenID 提供者管理的 URL：

```
http://openid.example.com/users/omjennyg
```

284 客户端通过向传入的它想查找的账户 URI 发起 GET 请求，构造了一个 WebFinger 请求：

```
GET /.well-known/webfinger?resource=acct%3Ajenny%40example.com HTTP/1.1
```

服务器应该响应一份该用户账户的 JRD 描述。该描述应该会包含一个叫作 `subject` 的附加 JSON 属性，它给出了被描述资源的 URI：

```
HTTP/1.1 200 OK
Content-Type: application/jrd+json

{
  "subject": "acct:jenny@example.com",
  "properties": {
    "http://schema.org/name": "Jenny Gallegos",
    "http://schema.org/email": "jenny@example.com"
  }
}
```

确实就是这样。JRD 文件格式参与了大部分的工作，而 `acct: URI` 模式也几乎参与了每一件事。唯一 WebFinger 独有的东西就是 `subject` 属性和 Well-Known URI template：`/.well-known/webfinger?resource={uri}`。

这是一个完美的例子，展示了资源描述较资源表述工作得更好的这种情况。当有人你的网站上注册账户的时候，你可能会根据她的邮箱地址或 OpenID URL 来标识她。一个 OpenID URL 具有一个表述，但是你很可能无法控制 OpenID 服务器，所以你不能改变它的表述。一个邮箱地址完全没有表述。尽管如此，WebFinger 让你可以发布你的用户的账号描述。你可以通过注解对应的 `acct: URI` 来说明任何你想对这些账户信息进行的说明。

注4 定义见互联网草案“draft-ietf-appsawg-acct-uri”，我之前曾提到过。

本体动物园 (Ontology Zoo)

这是第 10 章中“语义动物园”的续集，罗列了一些感兴趣的 RDF Schema 词汇表。我们有着大量的 RDF Schema 词汇表，但是它们分散在整个互联网上。我只想聊聊两个流行的词汇表以及一个收集词汇表的网站。

我专注于那些在面向消费者时比较有用的 API。大部分真正重型的词汇表都是用于科研或医疗应用的，它们并不描述那些由 Web 提供的文档的语义。你可以从 SWEET 本体 (<http://sweet.jpl.nasa.gov/>) 中检出设计专为科研使用的超大词汇表。

schema.org RDF

- 网站：Schema 主页 (<http://schema.org/>)
- 词汇表文档：见 http://schema.org/docs/schema_org_rdfa.html
- 语义：人类可能会在线搜索的一些事物类型

在本书的前面部分，我介绍了一些定义在 schema.org 上的一些概念，比如 Person、CreativeWork 等，并将它们作为 HTML 微数据项。这也正是 schema.org 网站上对它们的介绍。但是在幕后，这些概念都是由一个机器可读的 RDF 词汇表定义的。这就是我在本章中一直在引用的词汇表，它使用了例如 <http://schema.org/Person> 和 <http://schema.org/birthDate> 这样的 URI。我使用了相同的 RDF 词汇表来为我的 alps.io 网站生成所有 schema.org 的微数据项的 ALPS 版本。

如果你正在使用 RDF 或 JSON-LD，你可以在描述资源的时候引用 schema.org RDF 词汇表。这可以让你采用描述策略来替代表述策略，从而能谈论所有 schema.org 可以谈论的内容。

将 schema.org 元数据在资源的 HTML 表述中的作用……

```
<div itemscope itemtype="http://schema.org/Person">
  <span itemprop="birthDate">1987-08-25</span>
</div>
```

……与 schema.org 的 RDF 词汇表在描述没有表述的资源时的作用进行比较：

```
@prefix schema: <http://schema.org/>
<acct:omjennyg@example.com> a schema:Person ;
                             schema:birthDate "1987-08-25" .
```

当然，你并不只是被限于描述人。只要在 schema.org 上看到那些跟你所做方式相同的概念，你便可以使用由 schema.org RDF 词汇表描述的接近 1000 的概念中的任何一个。

FOAF

- 网站：FOAF 词汇表规范页面 (<http://xmlns.com/foaf/spec/>)
- 词汇表文档：可以从 <http://xmlns.com/foaf/spec/index.rdf> 下载
- 语义：人和组织

FOAF 是最著名的 RDF Schema 本体。这是一个用于描述人、组织和他们之间关系的非正式的工业标准。下面是如何使用 FOAF 词汇表，并采用 RDF/Turtle 表示一个人的名字和他的生日^{注5}的一个例子：

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
<acct:omjennyg@example.com> a foaf:Person ;
                                foaf:name "Jennifer Gallegos" ;
                                foaf:birthday "08-25" .
```

vocab.org

- 网站：见 <http://purl.org/vocab/>
- 词汇表文档：有多种多样的
- 语义：多种多样的

这是一个托管了 RDF Schema 文档的网站，由 Ian Davis 维护。与我自己的针对 ALPS 文档的 alps.io 注册表类似。该集合是五花八门的，包括了我早先在脚注处提过的 BIO 词汇表，还有一个用于描述各类威士忌酒的词汇表。

作为一个政策问题，vocab.org 同样也允许任何人宣称使用以 <http://vocab.org/vnd/> 起首的命名空间 URI。这意味着我可以使用 URI <http://vocab.org/vnd/mamund.com/2013/my-wonderful-resource> 发布一个描述某资源的 RDF 文档。我并不能控制 vocab.org，并且我也不能向它上传文件，所以该资源也永远不会具有表述。但是我拥有该 URI，并且我可以随意地对它进行描述。

总结：描述策略生机盎然！

当你的思维习惯了 REST 之后，RDF 文档会让你觉得有些陌生。有一部分的原因是因为我们拥有很多不同的方式来编写“RDF 文档”，但是大部分原因是因为 RDF 文档通常都忽略了 Fielding 约束。存在着真正的 RDF 文档，并且有着真实的人在使用它们完成工作，这些文档包含的 URL 会在响应你的 HTTP GET 请求时给你 404 错误。以至于表述策略的 REST 的观点看来，这些“并不存在的”URL 和包含它们的文档是损坏的。理解了描述策略，你才会更加理解这些事物的意义。

注5 不只是出生的日期！此处更加复杂。

如果可以忽略描述策略的话，这一章以及在日常生活中都会变得更加简单。多亏有了 Linked Data，你们就可以这样做了。Linked Data 运动表明更好的做法是以一种满足 Fielding 约束的方式来使用 RDF。只需要在 Web 上发布你的 RDF Schema 词汇表，并确保你只使用它来描述那些同样也存在于 Web 的资源即可。

Linked Data 的成长空间是非常大的。当采用机器可读的方式来描述应用语义时，RDF Schema 和 OWL 比 ALPS 更为强大。而你无须放弃 Fielding 约束便可以利用到这些技术的优势。

◀ 287

但是我不能假装 Linked Data 就是故事的全部。语义网比 Linked Data 更加古老，尽管这样，不是每个人都在赶 Linked Data 的潮流。当你在使用语义网技术的时候，你将会遇到大量文档，而这些文档中的 URL 被证明其实都是 URI。我不认为你应该创建更多的这种文档，但是在处理已经存在的这类文档时，你需要理解它们是什么意思。它们都是那些不具备表述的资源的描述。

CoAP:嵌入式系统的REST

受限应用协议 (Constrained Application Protocol) ^{注1} 是一种专门设计用于低功耗嵌入式环境比如家庭自动化系统的协议。CoAP 是受 HTTP 启发的并且能够用于发布超媒体驱动的 RESTful API, 但是它是一种非常不同于 HTTP 的协议。CoAP 给高度受限的环境: 一个由许多小型的、廉价的计算机组成并通过低容量的网络进行通信的“物联网”带来了 web 样式的架构。

CoAP 是设计用来适应在电力消耗、网络带宽以及处理能力等方面苛刻的局限性的。它的世界更像是 20 世纪 70 年代的 ARPAnet, 而非现在人们所使用的网络。CoAP 请求和响应都非常小。在一个通过家庭输电线组成的网络上, 一条 CoAP 消息不应该大于 1024 字节。在低功耗的无线网络中, 你很可能不会想要数据超过 80 字节。

但是在网络布局上, 这些环境和万维网是很类似的。这里并不存在单一的“API 提供方”来为许多类似的客户端提供服务。取而代之的是, 来自不同厂商的设备以一种看起来随机的方式被布置在同一个房间里。它们有的拥有数据可以提供, 有的有能力让现实世界发生一些事情。极少情况下, 它们中的某个设备才会可能要获得人类长时间的注意来回答是或者不是的问题。

这些设备必须通过网络互相定位、了解其他设备的功能和弄清楚如何与之一起工作, 而所有这些工作需要很少或者一点也不需要来自于那些安装该设备的人的指导。这真的很混乱。在这种彻底去中心化的环境中, 就像 Web 一样, 基于超媒体的策略是让它们能有机会正常工作的唯一选择。

注1 一个作为互联网草案还处于开发阶段的开放标准“draft-ietf-core-coap.”

CoAP请求

你应该很清楚一个标准的 HTTP 请求是如何工作的。客户端会打开一个连到服务器的 TCP 连接 (connection)，发送请求，然后在相同的连接上等待某个响应。CoAP 是设计为运行在 UDP 上的，UDP 是 TCP 的一个姊妹协议，它并不支持连接。一个 CoAP 客户端在向服务器发送一个请求消息之后，就着手做别的事情去了。客户端并不知道响应消息（如果存在的话）会在什么时候到达。

下面是一个遵循 CoAP 标准的请求消息示例：

```
CON [0xbc90]
GET /temperature
(Token 0x71)
```



我需要说明的是这不是一条真正的 CoAP 消息。这是一条人类可读版本的消息。我尽可能将其格式化来使得它看起来和 HTTP 相似。这条实际上通过 UDP 发送的消息被打包成人类无法理解的二进制格式：CON 变成二进制整数 00，GET 变成 0001，等等。

这个请求消息是什么意思呢？在 HTTP 中 GET /temperature 应该是对你有意义的。CoAP 定义了 4 种基本的方法 (GET、POST、PUT 和 DELETE)，尽管它们的语义和在 HTTP 中的语义有些许不同。

CON 代表 “Confirmable”，这意味着这条消息需要一条来自于服务器端的确认消息 (acknowledgment message)（我会在后面的章节中对此进行讨论）。

十六进制数 0xbc90 是 “消息 ID”，它将被用于确认消息中。没有了消息 ID，客户端在发送了两次 GET 请求并收到两个响应之后就不能知道哪个响应对应哪个请求了。

十六进制数 0x71 是一个 “令牌 (token)” 。一个 CoAP 可以触发多个响应，这个 token 会用在所有响应中，而不仅仅是最初的确认消息中。在确认消息之后发送的响应会拥有新的消息 ID，但是它们会通过令牌来和最初的请求关联起来。

CoAP响应

正如上文所述，客户端发送了请求之后，就着手做别的事情去了。但是它发送的消息是一个 CON 消息，它需要一份回执通知。最终，起初接收该消息的那个服务器将会把这个回执通知以应答消息 (acknowledgment (ACK) message) 的形式发送出去。

如下是一条人类可读的应答消息。

```
ACK [0xbc90]
2.05 Content
(Token: 0x71)
Content-Format: text/plain;charset=utf-8
22.5 C
```



再次声明，我将这条消息进行了格式化来处理来使得它像 HTTP。它并不是真的这种样式。真正的 CoAP 消息是被打包成紧凑的二进制格式的。比如媒体类型 application/json 是用八位二进制字符串 00110010 来表示的。

- “ACK”的意思是这条消息是对之前的某一条消息（我之前展示给你的那条 Message ID 为 0xbc90、Token 为 0x71 的 CON 消息）的一个确认收据 (acknowledging receipt)。
- 2.05 Content 是一个状态码，等价于 HTTP 的 200 OK。
- Content-Format 是 CoAP 的一个选项，它的用途和某个 HTTP 报头是相同的。Content-Format 选项做的就是 HTTP 的 Content-Type 报头的工作。
- 字符串 22.5 C 是负载 (payload)，在 HTTP 中它被称为“实体消息体 (entity-body)”。

请求和响应是完全不同的消息。它们不会像 HTTP 的请求和响应那样共用一个 TCP 连接。它们是通过消息里面的数据：消息 ID(0xbc90) 和 token (0x71) 来连接起来的。

消息种类

每个 CoAP 请求都和一个方法相关联。CoAP 定义了 4 种方法，它们都是以 HTTP 方法来命名的：GET、POST、PUT 和 DELETE。CoAP 方法并不完全和 HTTP 方法一致，但是它们具有相同的基本属性：GET 是安全的，PUT 是幂等的，等等。

CoAP 额外定义了一种协议语义——消息类型。在 CoAP 中，请求和响应是由不同的消息传递的，而消息类型就是用于解决这一问题的。每条消息都是下面这 4 种消息类型中的一种：

- 一条可确认的 (confirmable) 消息 (CON)，需要一条确认消息 (ACK)。客户端会一直发送 CON 消息直到它收到了一条拥有相同消息 ID 的 ACK 消息。
- 一条非可确认 (nonconfirmable (NON)) 消息不需要确认消息。只有安全的请求 (也就是 GET 请求) 可以被设定为可不确认的。
- 确认消息 (ACK) 用来确认之前的某条消息已经被接收并处理了。
- 相比之下，重置 (reset) 消息 (RST) 也是对之前某条消息的确认，但指的是接收端不能处理它。接收端可能重启了并丢失了必要的上下文信息，或者它可能由于网络不稳定丢失了之前的消息。

大致来说, 这些消息类型是在重现 HTTP 的请求 - 响应结构。一条 CON 消息加上一条 ACK 消息等价于一对 HTTP 的请求和响应。客户端在发送 CON 消息后如果并没有收到任何确认消息就应该再次发送这条 CON 消息, 就像一个 HTTP 客户端如果发送 GET 请求后没有收到任何响应就应该再次发送这个请求。

但是 CoAP 有两个有趣的功能, 是 HTTP 所没有的。一个是 (延迟响应) 依赖于一个 CoAP 请求能够触发多个响应的事实。另一个 (多播消息) 利用了 HTTP 所不能使用的功能, 因为 TCP 并不支持。

延迟响应 (Delayed Response)

假设客户端发送了一条需要花费很长时间来处理的 CON 消息。服务器可以迅速用一条 ACK 消息来作为响应。这就是告诉告诉客户端, 这个消息已经被接受了, 客户端可以停止发送 CON 消息了。客户端可以预期随后的第二条消息将包含真正的响应。

这第二个响应并不是一条 ACK 消息。它可以是一条 CON 消息, 也可以是一条 NON 消息。服务器就和客户端换了个位置。这第三条消息拥有一个消息 ID, 这个 ID 不同于客户端最初发送的 CON 消息的消息 ID, 但是这条消息复用了 token, 这样客户端就知道这是对当初的 CON 消息的响应了。

这个情况类似于从网上书店购书。你发送一条 CON 消息, 书店会迅速给你发送一封邮件收据 (一条 ACK 消息) 来确认你的购买行为。但是你买的书需要花上几天时间才能运到。你可能必须要签收快递 (用一条 ACK 消息来响应服务器发送的 CON 消息), 或者邮递员可能仅仅将书放在你的前廊 (这本书就是一条 NON 消息)。不管哪种方式, 你都会收到这本书以及一个单据 (token 令牌), 这个单据就将这本书与你最初的订单 (你最初的 CON 消息) 关联起来了。

HTTP 定义了一种响应码 (202, Accepted) 和上面的作用类似, 但是 HTTP 没有为服务器定义它处理完之前 “accepted” 的消息后, 再次与客户端通信的方法。到那时, TCP 连接早就已经关闭了。有一些办法可以避免这一状况的发生 (我将在附录 B 中讨论 Accepted 响应码时进行介绍), 但是并不存在一种定义明确的解决方案。而对于 CoAP 而言, 其解决方案是内建在其协议中的。

多播消息 (Multicast Message)

CoAP 客户端可以使用 UDP 多播 (multicast) 来将一条消息广播给本地网络中的所有机器。TCP 并不支持多播, 所以你不能在 HTTP 中使用它。

对 CoAP 广播(以及通常的 CoAP)而言,一个老套的用例就是家庭自动化。在这个场景中,你的温度调节器、冰箱、电视、电灯开关以及其他家用电器都内置有价格便宜的嵌入式处理器,它们通过一个低功耗的本地网络来进行通信。当你添加一件新的电器时,它会检测网络中其他的计算机、通过交换超媒体文档来了解它们的功能并开始与之协同工作。

这使得你的电器可以在没有你的直接介入的情况下协调它们自己的行为。当你打开烤箱时,温度控制器能够注意到这一事件并调低厨房的温度。你可以拿出你的手机,得到你当前所在房间的所有灯的列表,然后使用你的手机来调低灯的亮度,而不再需要走到开关处操作。

这种家庭自动化的用例已经存在超过 50 年了,我个人认为这是一个很俗气的白日梦,但是也还有一些其他的场景,多播发现(multicast discovery)在这里面就不那么俗气了。多播可以让一台手机与房间内的所有其他手机通话。它可以让一组科研仪器共享仪表读数,或者将一些低带宽的无线外围设备连接到桌面计算机上。

任何时候,只要在同一地方有许多小型的计算机,CoAP 和 UDP 多播就可以让它们互相发现对方并确定如何与之协作。这其中起作用的原则就是无状态性、可寻址性以及自描述消息,也就是 REST 的核心原则。

CoRE Link Format

- 媒体类型: application/link-format
- 出处: RFC 6690
- 媒介: 纯文本
- 协议语义: 使用 GET 进行浏览和查询
- 应用语义: 无

当然,REST 并不是传输协议。REST 通过交换超媒体表述来工作,而这些超媒体表述常常都非常大。当你全部的响应必须限定在 80 或者 1024 个字节内的时候,你就不能发送 HTML 或者 Collection+JSON 表述了。^{注 2}

HTTP 可以压缩表述来节省带宽,但是压缩在这里并不管用。一个光传感器并没有足够的处理能力来在合理的时间内解压缩一个表述,更不用说解析一个解压缩过的文档了。它可能没有足够的内存来保存完整的文档。这就是为什么 CoAP 的开发人员为嵌入式应用专门设计了一种新的超媒体格式: CoRE Link Format。

◀ 294

注 2 互联网草案“draft-ietf-core-block”将允许把一个大的表述通过若干个 CoAP 消息进行分解。这有很大帮助,但是围绕这个特性来构建一个传统的 web 风格的 API 将是非常没有效率的。

如下是第 5 章中的迷宫游戏中的某个单元格的 CoRE Link Format 表述。

```
</cells/M>;rel="current";rt="http://alps.io/example/maze#cell",  
</cells/N>;rel="east";rt="http://alps.io/example/maze#cell",  
</cells/L>;rel="west";rt="http://alps.io/example/maze#cell",  
<http://alps.io/example/maze>;rel="profile"
```

这其中有 4 个链接，它们都采用 rel 属性。其中 3 个链接 (current、west 和 east) 都分别指向迷宫中的单元格 (cell)。第四个链接 (profile) 指向某个用于说明 current、east、west 以及 cell 的含义的 profile。

不同于 CoAP 消息，CoRE LINK Format 格式的文档是人类可读的。它不是一种二进制格式。我对这个表述所做的唯一的修改就是在每个链接后添加了一个换行符，这样它更适合在页面展示。CoRE Link Format 可以使超媒体容纳到千字节，而完全不破坏人类可读性。

CoRE Link Format 使用和 HTTP 的 Link 报头相同的语法来描述一个链接。RFC 6690 定义了一些扩展参数，值得注意的有，包含一个 URI 的 rt 参数，这个 URI 表明了该链接的另一端的资源的抽象语义类型 (见第 12 章)：

```
</cells/N>;rel="east";rt="http://alps.io/example/maze#cell"
```

RFC6690 还提出了一些可选技术用于向 CoAP 资源发送查询请求和接收 CoRE Link Format 格式的响应。这些技术为 application/link-format 媒体类型提供了一些功能，这些功能类似于 Collection+JSON、OData 或者其他一些集合模式的实现方案的功能。

但是在 CoRE Link Format 文档中没有位置来记录数据。CoRE Link Format 是一种最纯粹的超媒体。它只能表示状态转换。真正的数据——比如设备读数、统计结果和人类可读的消息——都必须以一些其他的数据格式来提供，比如 JSON。

295 结论：非HTTP协议的REST

CoAP 和 HTTP 有很大不同，但是它们的架构都是 RESTful 的。CoAP 系统遵循了无状态的限制。事实上它比 HTTP 更加无状态，因为它的请求和响应并没有通过一个 TCP 连接来绑定起来。CoAP 定义了和 HTTP 类似 (而不相同) 的协议语义。CoAP Link Format 不能表示数据，但是它是一个真正的超媒体格式，而且它使用的超媒体控件要比 HAL 要多。

一台 CoAP 设备可以连接到网络上，发送一条 UDP 多播消息来查看周围有什么其他设备，开始检测这些设备都提供什么东西。这些都不需要任何人类交互。REST 的超媒体约束使得这样的灵活性成为可能，并且它是实现那些诸如家庭自动化之类的长远美梦的唯一

现实的解决方案。一台想要和你的微波炉通信的电冰箱不能对你所安装的微波炉的型号有所挑剔。

在我看来，就整体而言，这种情况就类似于 API 的世界。我们生活在一个房子（因特网）里，里面有数以千计的有用但是难以理解的可编程电器（API）。REST 就是要使这些电器在没有人类参与的情况下一起工作。

成功意味着在应用语义上达成了共识。电器和 API 在讨论相同的事物时都应该使用相同的单词。它也意味着宣传我们的协议语义。所有的“电器”都必须说明它所做的工作，这些信息不是要写进一个纸质的手册里，然后放到资料夹中等着上面落满灰尘，而是要以一种另一台“电器”能够明确理解的方式放到网上。这可能看起来有点疯狂，像是白日梦，但是这是管理我们所创造的复杂事物的唯一方式。

状态法典

HTTP 状态码是一个依附于 HTTP 响应的 3 位数字。它是协议语义的一部分，能在最基本的层面上让客户端知道服务器在尝试处理请求时发生了什么事情。由 HTTP 规范定义的 41 个 HTTP 响应码源自可以被任何 API 所使用的一组基本的协议语义。

除了 HTTP 重定向，以及尽管非常著名的“404 Not Found”错误页面之外，我们并没有真正地在万维网上将状态码用起来。人们是通过阅读作为响应的一部分的实体消息体来了解请求发生的状况的，而不是通过查找 HTTP 标准中的某个数字编码。当你在网站上填写表单时，如果正好忘了填写其中的一个必填项，那么当你提交之后服务器会返回一个错误消息，但是与该错误消息所关联的响应码却是 200 (OK)。

这很好，你甚至无须查看状态码。你阅读了错误消息并纠正了刚才的问题。但是如果一个 API 的行为也是以这样的方式进行表现的话，将会欺骗到它的客户端！计算机程序非常擅长查找数字编码，但是在理解普通的记叙性文体时却非常糟糕。当你在一个错误状态下提供 200 状态码时，你必须在你的 API 中编写额外的文档来进行说明，OK 并不是必定表示 OK。而额外的文档对你的用户而言却意味着更多的工作。

在 API 的世界里，HTTP 响应码则变得非常重要。它们告诉客户端如何看待实体消息体中的文档（到底是一份表述还是一段错误消息），或者在无法理解实体消息体时客户端应该做什么。客户端（或是介于服务器和客户端之间的中间组件，比如代理或防火墙）只需要通过查看响应的前几个字节就可以理解 HTTP 请求的运行状况。

即便如此，某些 HTTP 状态码几乎是没用的。而有一些也仅在非常有限的条件下才有用，还有一些状态码只有经过对细节非常仔细地审查后才能区分它们之间的差别。对那些习惯于万维网的人来说（即我们所有的人），各种状态码让人觉得眼花缭乱。

在这一篇附录中，我会为每个状态码都给出简明的说明，同时就何时应该在你的 API 中使用它们给予一些提示，并提出一些我个人的、关于状态码在 API 设计中的重要性的一些建议。如果一个客户端必须做一些特别的事情来获取某种状态码，我将会对它进行说明。我同样也会罗列出哪些 HTTP 响应报头和什么类型的实体消息体会随响应码一起由服务器进行发送。这篇附录是为 API 开发者准备的，但也是为那些接收到奇怪的响应码而不知其意的客户端作者而准备的。

与链接关系和媒体类型一样，IANA 维护了一个 HTTP 状态码的官方注册表。此处的“官方”基本上是指“被定义在某个 RFC 中”。在本篇附录中，我将会讨论在 RFC 2616 中提到的全部 41 个状态码，虽然它们中的某些（主要是那些与代理相关的）有一点超过了本书的范围。我同样也会讨论到一些定义在其他 RFC 中的状态码，尤其是 RFC 6585，其适当的名称是“附加的 HTTP 状态码”。

我将不会讨论 CoAP 的 HTTP-inspired 状态码（4.04 Not Found）以及由像 WebDAV 这类扩展所定义的 HTTP 状态码。同样我也不会讨论由 web 服务器实现引入的但没有在任何地方进行正式定义的状态码。这包括了 509 (Bandwidth Limit Exceeded) 以及 nginx 的多个内部错误报告码，比如 499 (Client Closed Request)。

问题细节文档

在第 10 章中，我提到过问题细节文档——给出了 HTTP 状态码的人类可读说明的一种简短的超媒体文档。请不要忘了这些！你可以使用它们来为一个像 400 (Bad Request) 这样的通用状态码添加 API 特定的细节信息。我们没有必要为了传达详细的错误信息而去发明一种新的表述格式（或者更糟的是发明一种新的状态码）。如果你的表述格式具有为错误报告而准备的属性槽（slot），就像 Collection+JSON 那样，你很可能就不再需要问题细节文档了。

记住，详细的错误报告并不能作为你在某些情况出现问题时提供 200 (OK) 的借口。这意味着你的表述必须始终与你的 HTTP 状态码保持一致。

状态码家族

HTTP 状态码的第一位数字是表明请求进展情况的一个非常通用的指示。HTTP 规范使用 1 到 5 作为起首数字分别定义了 5 个状态码家族。我将会按独立的部分逐个对它们进行介绍：

1xx: Informational

这些响应码仅在 HTTP 客户端与服务器之间进行协商时使用。

2xx: Successful

客户端所要求的任意的状态转换已经发生。

3xx: Redirection

客户端要求的状态转换没有发生。但是如果客户端愿意发起一个稍有不同的 HTTP 请求，该请求应该会完成客户端要求的行为。

4xx: Client Error

由于 HTTP 请求的问题，导致客户端要求的状态转换没有发生。该请求可能有缺陷、不合逻辑、自相矛盾或者该请求无法被服务器接受。

5xx: Server Error

由于服务器端的问题，导致客户端要求的状态转换没有发生。客户端或许什么都做不了，只能等待问题被修复。

四个状态码：最低限度

在我们开始遍历状态码的大型列表之前，我想罗列一下在我看来对于 API 来说最低限度的 4 个状态码。我们从每个状态码家族中选出了一个成员（除了 1xx，你或多或少可以忽略它）：

200 (OK)

一切都非常顺利。实体消息体中的文档（如果有）是某个资源的一份表述。

301 (Moved Permanently)

当客户端触发了某个将资源从一个 URL 移动到另一个 URL 的状态转换时将会发送该状态码。在移动之后，向老的 URL 发送的请求将同样会导致一个 301 状态码。

400 (Bad Request)

客户端存在问题。实体消息体中的文档（如果有）是一段错误消息。希望客户端能理解错误消息并使用它来修复问题。

500 (Internal Server Error)

服务器端存在问题。实体消息体中的文档（如果有）是一段错误消息。该错误消息可能帮不上什么忙，因为客户端不能修复服务器的问题。

如果我可以再多添加两个，它们会是不同类型的客户端错误：404 (Not Found) 和 409 (Conflict)。当你需要给予更为详细的信息时，你可以采用大型列表里的其他状态码，或者提供一个问题细节文档。

300 现在是时候来看看大型列表了。除非另外注明，所有这些状态码都正式定义在 RFC 2616 中。

1xx: Informational

1xx 响应码仅在 HTTP 客户端与服务器之间进行协商时使用。

100 (Continue)

重要性：低到中等。

这是应对 HTTP look-before-you-leap (LBYL) 请求的可能响应中的一种，我曾在第 11 章中描述过该请求。该状态码指示客户端应该重新发送它的初始请求，包括在首次请求中被省略了的（可能较大或较敏感的）表述。客户端不再需要担心发送表述后又被拒绝的问题。应对 look-before-you-leap 请求的另一个可能的响应是 417 (Expectation Failed)。

请求报头：为了发起一个 LBYL 请求，客户端必须将 Expect 报头设置为“100-continue”这一字面量值。客户端同样必须设置任何服务器所需要的其他报头，服务器将使用这些报头来决定是响应 100 还是 417。

101 (Switching Protocols)

重要性：非常低，潜在为中等。

客户端只会在当请求中使用了 Upgrade 报头来通知服务器它想要选择使用除 HTTP 之外的别的协议时才会收到该响应码。状态码为 101 的响应的意思是说“没问题，现在我开始使用另一种协议跟你交谈”。通常，一个 HTTP 客户端会在读完来自服务器端的响应之后立刻关闭 TCP 连接。但是响应码为 101 意味着是时候让客户端在保持连接打开的情况下，结束作为 HTTP 客户端的角色，并同时开始充当起某些别的类型的客户端。

我们很少使用 **Upgrade** 报头，虽然它可以用于将 HTTP 升级到 HTTPS，或是将 1.1 版的 HTTP 升级到最终即将到来的 2.0 版。它同样也可以被用于将 HTTP 切换到一个像 IRC 这样完全不同的协议，但是这要求 web 服务器同样也能成为一台 IRC 服务器，以及 web 客户端同样也可以成为一个 IRC 客户端，因为这样一来服务器将立即开始在同一个 TCP 连接上使用新的协议。

请求报头：客户端将 **Upgrade** 设置为一组相比于 HTTP 协议，它更加想要选择使用的协议。

响应报头：如果服务器想要升级，它会发送回 **Upgrade** 报头来表明它所选择切换的协议，后面紧跟一个空白行。相比于原来关闭 TCP 连接，现在服务器会开始启用新的协议，并会一直使用新的协议直到连接关闭。

2xx: Successful

2xx 状态码指示客户端所要求的任意的状态转换已经发生。

200 (OK)

重要性：非常高。

大部分情况下，这是客户端希望看到的状态码。它指示状态转换已经结束，并且在 2xx 系列中找不到更加具体且合适的状态码的时候可以使用它。

实体消息体：对于一个 GET 请求来说，是作为 GET 目标的资源表述（这将引起应用状态的改变）。对于其他类型的请求来说，是针对资源状态改变的描述：即被选择资源当前状态的表述，或者是对状态转换本身的描述。

201 (Created)

重要性：高。

当服务器基于客户端的请求创建了新的资源后它会发送该状态码。

响应报头：**Location** 报头应该会包含新资源的标准 URL（canonical URL）。

实体消息体：应该描述并链接到新创建的资源。如果你使用 **Location** 报头来告诉客户端资源事实上位于哪里，那么说明该资源的表述是可被接受的。

202 (Accepted)

重要性：中等。

客户端的请求不能或不会被实时地处理，并且将会在后续被处理。该请求表面上看上去可能是有效的，但是当服务器在实际处理它时可能会发现该请求其实是存在问题的。

当一个请求触发了某个异步动作（某个现实世界中的动作）时，或者是某个状态转换花费了过多的时间以至于没有必要让客户端一直等待该响应时，包含了该状态码的响应便是一种合适的选择。

请求报头：当客户端不希望得到 202 状态码，而想要获取真正的响应时，它可以通过设置 **Prefer** 报头（见附录 B）来告诉服务器它将会愿意等待的时间。

302 响应报头：尚未完成的请求应该要暴露某些类型的资源从而让客户端可以在稍后进行对结果的复查。而 **Location** 报头便可以包含链接到该资源的 URL。

实体消息体：如果无法让客户端在稍后来复查请求，那至少应该给出一个对请求何时会被处理的预估。问题细节文档也许适合该场景，虽然严格意义上来说这并不是一个“问题”。

Retry-After：服务器对于何时能准备好全部的响应会有一个估算时间，我们可以使用 **Retry-After** 报头来指示该估算时间。该报头被设计与 5xx 和 3xx 响应码一起使用，但是也可以被安全地用于 202 响应。

203 (Non-Authoritative Information)

重要性：非常低。

该状态码与 200 (OK) 相同，但是除此之外服务器还想让客户端了解一些并非来自该服务器的响应报头。这些报头可能反映自客户端的前一个请求，或者是从第三方组织获得的。

响应报头：客户端应该要知道：某些报头可能并不准确，而其他被传递过来的报头连服务器也不知道它们是什么意思。

204 (No Content)

重要性：高。

该状态码通常在响应例如 **PUT** 请求这样的非安全请求时被发送。它的意思是服务器已经执行了状态转换，但是它拒绝发送回任何表述或状态转换的描述。

服务器可能会在对应 GET 请求的响应中发送回 204。这意味着该请求的资源存在，但是它拥有一个空的表述。相比于 304 (Not Modified)，204 通常见于浏览器中的 JavaScript 应用。它让服务器可以告诉客户端，客户端的输入已经被接受，但是该客户端不应该改变任何的 UI 元素。

实体消息体：不允许。

205 (Reset Content)

重要性：低。

该状态码类似于 204 (No Content)，但是它暗示了客户端应该重置数据来源的视图或数据结构。如果你在你的 web 浏览器中提交了一个 HTML 表单，并得到了 204 (“No Content”) 的响应，那么你的数据仍然还在表单里，而你还可以修改它们。但是如果你得到的是一个 205，这些表单字段将会重置为它们的原始值。从数据条目的角度来说：204 有利于对单条记录进行一系列的编辑；而 205 则有利于连续输入一系列记录的场景。

◀ 303

实体消息体：不允许。

206 (Partial Content)

重要性：对于支持局部 GET (partial GET) 的 API 来说非常高，而对于其他 API 则比较低。

该响应码类似于 200 (OK)，但是它被指定作为局部 GET (partial GET) 请求的响应：也就是说，该请求使用了 Content-Range 这一请求报头。客户端通常会发起一个局部请求来恢复一个被中断的大型二进制表述的下载。我在第 11 章中讨论了局部 GET。

请求报头：客户端发送了一个用于设置 Content-Range 报头的值。

响应报头：Date 报头是必需的。而 ETag 和 Content-Location 报头应该被设置成与“早前和表述作为整体一起被发送回客户端的相应报头”相同的值。

如果实体消息体是表述中的某个单个字节范围 (single byte range)，那么响应作为一个整体还必须具有一个 Content-Range 报头，用于说明当前提供的是哪些字节。如果消息体是一个 multipart 实体（这就是说，提供的是表述的多个字节范围，即 multiple byte ranges of the representation），则实体消息体整体的媒体类型是 multipart/byteranges，而其中每个部分必须具有它们自己的 Content-Range 报头。

实体消息体：将不会包含全部的表述，只有表述中的一个或多个字节序列。

3xx: Redirection

客户端要求的状态转换没有发生。但是如果客户端愿意再发起一个略有不同的 HTTP 请求，该请求应该会完成客户端要求的行为。通常，客户端需要向另一个不同的资源重复它的请求。

这是响应码中最复杂的一个集合，因为 301 (Moved Permanently)、302 (Found)、303 (See Other) 和 307 (Temporary Redirect) 都非常相似。很多应用在使用这些状态码时非常随意，在这种方式下，客户端就好比是一个经过超媒体弹球游戏机的小球一样，随意地在不同的 UR 间弹跳着，但是却没人考虑过这意味着怎样的应用语义。我在本节中的目标就是清除掉这些令人困惑的疑团。

300 (Multiple Choices)

重要性：低。

304 ➤ 当被请求资源具有多个表述时，服务器会在不知道客户端想要哪一个表述时发送该状态码。导致这样的原因要么是因为客户端没有使用 `Accept-*` 报头来指定表述，要么是因为它所请求的表述并不存在。

在这种情况下，服务器可以挑选一个它偏好的表述，然后将它和 200 (OK) 状态码一同发送回来。但是它也可能会决定发送 300 以及一组链接到不同表述的 URI。

响应报头：如果服务器拥有一个偏好的表述，它可以将链接到该表述的 URI 放入 `Location`。和大部分其他 3xx 状态码一样，客户端将会自动访问 `Location` 中的 URI。

实体消息体：一组超媒体链接，以及必要的应用语义，从而让用户可以在它们之间做出选择。

301 (Moved Permanently)

重要性：中等。

服务器知道客户端尝试访问哪个资源，但是客户端并不关心它用于请求资源的 URL。它希望客户端能注意到新的 URL 并在将来的请求中使用新的 URL。

你可以在 API 改变了 URL 结构之后使用该状态码来保持老 URL 的可用性，使其不被破坏。

响应报头：服务器应该将标准 URL 放入 `Location`。

实体消息体：服务器应该发送一个链接到新位置的超媒体文档。

302 (Found)

重要性：需要大家对该状态码进行重点了解，尤其是在编写客户端的时候。但我并不推荐使用它。

该状态码便是大部分与重定向相关的困惑的最终来源。它的处理方式本应该像 307 (Temporary Redirect) 那样。而事实上，它在 HTTP 1.0 中名字是 Moved Temporarily。不幸的是，在现实生活中，大部分客户端是像处理 303 (See Other) 一样来处理 302 的。与 303 的区别之处在于当客户端在 PUT、POST 或 DELETE 请求的响应中得到 302 响应码之后应该会做些什么。如果你对细节感兴趣，请查看下面 307 和 308 (Permanent Redirect) 对应的条目介绍。

为了解决歧义，在 HTTP 1.1 中，该响应码已经被重命名为 Found，而同时创建了一个新的响应码 307。但是 302 响应码仍然被广泛使用着，而它是具有歧义的，所以我建议你的服务器应该通过发送 303、307 和 308 来取代 302。

响应报头：Location 报头包含了客户端应当重新向其提交请求的 URL。

实体消息体：跟 301 一样，应该包含一个链接到新 URL 的超媒体链接。

305

303 (See Other)

重要性：高。

请求已经被处理，但是服务器没有发送响应文档，取而代之的是发送给客户端一个响应文档的 URL。这可能是一个静态的状态消息或者是某些更有意思的资源的 URL。在后一种情况中，303 是服务器在向客户端发送了资源的表述之后却不强制客户端下载所有数据的一种方式。客户端被预期使用 GET 请求来访问 Location 中提到的 URL，但是这并不是必需的。

303 状态码是可以用于对资源进行标准化的一种很好的方式。它可以使你的资源能通过很多 URL 被访问到，但是每个表述只有一个“真正的”URL。其他所有的 URL 都是使用 303 来链接到表述的标准 URL。举个例子，303 可以将一个发送到 `http://www.example.com/software/current.tar.gz` 的请求重定向到 `URL:http://www.example.com/software/1.0.2.tar.gz`。

可与 307 (Temporary Redirect) 进行对照理解。

响应报头：Location 报头包含了表述的 URL。

实体消息体：跟 301 一样，应该包含一个链接到新 URL 的超媒体链接。

304 (Not Modified)

重要性：高。

该状态码与 204 (No Content) 相似，在该响应中，消息体一定是空的。但是 204 是当没有消息体数据可以发送时使用的，而 304 是原本有数据，但客户端已经拥有了该数据时使用的，此时没有必要重新发送数据。

该状态码是与带有条件的 HTTP 请求同时使用的。如果客户端发送了一个 **If-Modified-Since** 报头以及值为 Sunday 的日期值，并且表述自 Sunday 之后就没有再发生过改变，此时 304 是非常合适的。使用 200 (OK) 也是合适的，但是再次发送表述将会浪费带宽，因为客户端应经拥有该表述了。

响应报头：**Date** 报头是必需的。如果早前请求的响应码是 200 (OK)，那么响应中的 **ETag** 和 **Content-Location** 报头应该被设置成与“早前发送回客户端的相应报头”相同的值。

如果缓存相关的报头：**Expires**、**Cache-Control** 和 **Vary** 在发送前已经被修改，那么它们是必需的。

306 关于这方面有着非常复杂的缓存规则，而我将不会在本书中谈到，但是服务器可以只发送更新了报头，而不带新的消息体。在一个表述的元数据发生改变而实体消息体没有变化时，这是非常有用的。

实体消息体：不允许。

305 (Use Proxy)

重要性：低。

该状态码用于告知客户端应该重复它的请求，但是是通过一个 HTTP 代理而不是直接发向服务器。该状态码很少被使用，因为服务器很少会关心客户端是否使用了特定代理。

该状态码在基于代理的镜像站点中会用得比较频繁。今天，一个 <http://www.example.com/> 的镜像站点会在一个不同的 URL 提供相同的内容，比如叫作 <http://www.example.com.mysite.com/>。而原始站点可以使用 307 (Temporary Redirect) 状态码来将客户端传送到合适的镜像站点。

如果是基于代理的镜像站点，你可以使用跟原始站点相同的 URL (<http://www.example.com/>) 来访问镜像，并将 <http://proxy.mysite.com/> 设置成为你的代理。此时，原始的 [example.com](http://www.example.com/) 可以使用 305 状态码来将客户端们路由到一个地理上更加靠近它们的镜像代理。

Web 浏览器通常不会正确地处理该状态码：这也是它缺乏人气的另一个原因。

响应报头：Location 报头包含了链接到代理的 URL。

306: Unused

重要性：无。

306 状态码从未被记录到 RFC 中。它是在互联网草案“draft-cohen-http-305-306-responses”中作为切换代理（Switch Proxy）之用描述的，代理服务器发送该状态码来让客户端开始使用另一个不同的代理。该互联网草案在 1996 年就已经过时，所以我们无须为它费神。

307 (Temporary Redirect)

重要性：高。

该请求尚未被处理，因为请求的资源并不在本 URL 内：它位于某个其他的 URL 上。客户端应该向另一个 URL 重新提交请求。

当对应 GET 请求时，请求的唯一内容就是让服务器发送一个表述，该状态码此时与 303 (See Other) 相同。一个典型的场景是当服务器想要将发起 GET 请求的客户端传送到一个镜像站点时，307 将是响应的很好选择。但是对应 POST、PUT 和 DELETE 请求，服务器预期会在对应请求的响应中采取一些动作，这是该状态码与 303 显著的区别。

◀ 307

对应 POST、PUT 或 DELETE 的 303 响应意味着操作已经成功，但是在对应该请求的本次响应中不会发送实体消息体。如果客户端想要响应的实体消息体，它需要向另一个 URL 发起 GET 请求。

对应 POST、PUT 或 DELETE 的 307 响应意味着服务器甚至尚未尝试执行操作。客户端需要向 Location 报头中的 URL 重新提交整个请求。

打个比方也许会对你有所帮助。你带着处方去某个药房配药。303 就好比药剂师告诉你“我们已经给你配好了药，请到下一个窗口领取你的药品吧”。而 307 则好比药剂师告诉你“我们药品已经卖光了，去隔壁药店看看吧”。

响应报头：Location 报头包含了客户端应该重新向其提交请求的 URL。

实体消息体：跟 301 一样，应该包含一个链接到新 URL 的超媒体链接。

308 (Permanent Redirect)

重要性：中等。

定义方式：互联网草案“draft-reschke-http-status-308”。

对应 GET 请求的 308 响应与 301 (Moved Permanently) 相似。但是对应非安全请求的 308 响应工作方式类似于 307 (Temporary Redirect)：客户端应该向 Location 报头中给出的 URL 重新提交请求。不同之处在于客户端应该在未来它想要发起的任何请求中继续使用新的 URL。

我们继续 307 (Temporary Redirect) 讨论中的药房类比，308 响应码好比药房已经停业，下次再来也无济于事。你需要带着你的处方，以及未来任何有需要买药时都去隔壁的药房买药。

该状态码定义在 HTTP 的一个扩展中，目前仍然以互联网草案的形式存在。从更加安全的角度考虑，即使在它成为 RFC 之后，我们或许还是会使用 307 来满足永久重定向。客户端很可能并不理解 308 响应码是什么意思。

4xx: Client-Side Error

该状态码指示在客户端存在着问题。问题可能与身份认证、表述格式、请求的时间或 HTTP 客户端本身有关。客户端需要在自己这端将问题修复。

308 问题细节文档（见第 10 章）对于 4xx 系列的状态码来说最为有用。对于大多数这种错误码来说，我认为实体消息体会包含一个“文档”。除非你使用了一种内建了错误报告机制的表述格式，不然我建议你将该文档整成一份问题细节文档。

400 (Bad Request)

重要性：非常高。

这是通用的客户端错误状态，当在其他 4xx 错误码中找不出更合适的选择时可以选择该错误码。通常在客户端通过 PUT 或 POST 请求提交表述，并且表述的格式正确，但是表述本身却没有任何意义时，服务器会使用该状态码。

实体消息体：可能会包含一个从服务器视角来说明客户端问题的文档。

401 (Unauthorized)

重要性：高。

客户端在没有提供适当的身份认证凭证的时候向受保护的资源发送请求。它可能提供了错误的凭证或完全没有提供凭证。凭证可以是用户名和密码、一个 API key 或者是一个认证的 token——任何 API 质询时所期望的内容。通常客户端向 URL 发起请求会接收到一个 401 响应，这样它就知道了该发送什么类型的凭证并采用什么样的格式。事实上，HTTP Digest 模式的认证便依赖于这种行为。

如果服务器并不想向未授权的用户承认资源的存在，它可以选择撒谎并发送 404 (Not Found) 来取代 401。这样做的副作用是客户端需要提前知道服务器期望对该资源进行什么类型的认证。像 HTTP Digest 这样的协议将无法正常工作。

响应报头：WWW-Authenticate 报头描述了服务器将会接受什么类型的认证。

实体消息体：文档描述了失败；为什么凭证（如果提供了）被拒绝，以及什么样的凭证将被接受。如果一个人类终端用户可以通过在网站上注册来获得凭证，或创建一个“用户账户”资源，那么一个链接到注册资源的超媒体链接也将会很有用。

402 (Payment Required)

重要性：无。

除了它的名字之外，该状态码并没有在 HTTP 标准中进行定义：它是“保留以备将来之用”的。这是因为目前还没有针对 HTTP 的小额支付系统。这就是说，如果可能出现 HTTP 的小额支付系统，那么 API 将会在这些系统出现的地方首当其冲。如果你想要通过 HTTP 请求向你的用户收费，你和他们之间的关系将使得这一点成为可能，而此时你将可以使用该状态码。

309

但是已经存在大量通过请求进行收费的 API，而我并不知道有任何这方面的 API 使用了该状态码。它将可能永远保持“保留”的状态。

403 (Forbidden)

重要性：中等。

客户端的请求格式正确，但是服务器并不想去执行它。不仅仅是因为凭证不足：如果是这样可以使用 401 (Unauthorized)。这更像是资源仅允许在特定时间或来自特定 IP 段的请求进行访问。

403 响应意味着客户端向其发起请求的资源真正存在。和 401 (Unauthorized) 一样, 如果服务器甚至连这些信息也不想提供时, 它可以选择撒谎并发送一个 404 (Not Found) 取而代之。

如果客户端的请求格式良好, 为什么是使用 4xx 系列 (客户端错误) 而非使用 5xx 系列 (服务端错误) 的状态码呢? 因为服务器是基于请求的多个方面来做出决策的, 并非只看它的格式: 比如对于请求发起时间在一天中不同时间段的限制。

实体消息体: 一个可选的用于说明请求为什么被拒绝的文档。

404 (Not Found)

重要性: 高。

404 大概算得上是最著名的 HTTP 状态码了。404 指示了服务器无法将客户端请求的 URL 映射到任何资源。我们来将它与状态码 410 (Gone) 进行对比, 这样会更加有帮助。请记住, 404 是可以用于掩盖 403 或 401 的一个谎言。有可能资源是存在的, 但是服务器并不想让客户端知道这一真相。

实体消息体: 一个可选的用于说明错误的文档。文档可能包含了一个用于在此处创建资源的超媒体控件 (或许使用的是 HTTP PUT)。

405 (Method Not Allowed)

重要性: 中等。

客户端尝试使用的某个 HTTP 方法对应的资源并不支持。举个例子, 一个只读资源仅可以支持 GET 和 HEAD。而集合资源通常允许 GET 和 POST, 但是并不支持 PUT 或 DELETE。

310 响应报头: Allow 报头会罗列出该资源支持的所有 HTTP 方法。下面是该报头的一个例子:

```
Allow: GET, POST
```

406 (Not Acceptable)

重要性: 中等。

当客户端对可接受的表述做了很多约束时 (可能会使用 Accept - * 请求报头), 服务器会在无法发送任何表述时发送该响应码。服务器也可以选择忽略客户端的挑剔, 简单地

发送响应码 200 (OK) 以及自己偏好的表述。这通常发生在人类的 web 上。

实体消息体：一个链接到可接受表述的超媒体文档，使用了一种与 300 (Multiple Choices) 中描述相似的格式。

407 (Proxy Authentication Required)

重要性：低。

你只能从 HTTP 代理看到该状态码。它跟 401 (Unauthorized) 很像，除了对应的问题不再是你在使用 API 时没有提供凭证，而是在使用代理时没有提供凭证。跟 401 一样，该问题可能是因为客户端没有提供凭证，或者提供的凭证是损坏或不足的。

请求报头：客户端使用 Proxy-Authorization 报头而非 Authorization 报头来向代理发送凭证。它的格式与 Authorization 是相同的。

响应报头：代理不再使用 Authenticate 报头，取而代之的是在 Proxy-Authenticate 报头中填充关于它所期望的认证类型信息。它的格式与 Authenticate 是相同的。

请注意，代理和 API 都可能需要凭证，所以客户端需要澄清 407 与 401 (Unauthorized) 之间的区别。

实体消息体：一个描述了失败的文档，就跟我在状态码 401 中所描述的一样。

408 (Request Timeout)

重要性：低。

如果 HTTP 客户端打开了一个与服务器之间的连接。但是从不发送请求（或者从不发送表示请求结束的空行），服务器应该最终发送一个 408 响应码来关闭这个连接。

409 (Conflict)

重要性：非常高。

客户端尝试在服务器上创建一个不可能或不一致的资源状态。什么是“不可能”或“不一致”取决于 API 的应用语义。一个基于集合的 API 会允许客户端 DELETE 一个空的集合，但是当客户端尝试 DELETE 一个还包含成员的集合时，将会发送 409。

响应报头：如果该冲突是由于某些其他资源的存在（比如，客户端尝试创建的某个特定的资源已经存在了）而造成的话，那么 Location 报头应该链接到该资源的 URL：也就

是说，冲突的来源。

实体消息体：应该包含了一个描述了冲突的文档，这样客户端才有可能解决它们。

410 (Gone)

重要性：中等。

该响应码很像 404 (Not Found)，但是它提供了更为更多的信息。当服务器知道所请求的 URL 曾经指向某个资源，但目前已经不再指向该资源时，就会使用该状态码。服务器并不知道该资源的任何新的 URL；如果它知道，它可以发送 301 (Permanent Redirect)。

与 301 永久重定向相同的是，410 响应码暗示了客户端应该从它当前的词汇表里去除掉当前的 URL，并停止向该 URL 继续发送请求。与 301 永久重定向不同的是，410 没有为毁坏的 URL 提供替代；它只是提示不存在了。RFC 2616 建议为“那些限时、增值的服务以及那些属于某些不再工作于该服务器站点的个体的资源”使用 410 响应码。

你可能会想使用这个响应码来响应那些成功的 DELETE 请求，但是这有点矫揉造作了。客户端在它没有发起请求前不会知道资源是被它删除了还是已经不存在了。正确的做法是使用 200 (OK) 来响应一个成功的 DELETE 请求。

411 (Length Required)

重要性：低到中等。

包含了表述的 HTTP 请求应该将 Content-Length 请求报头设置为实体消息体的长度(以字节为单位)。某些情况下这对客户端来说是非常不便的：举个例子，当表述是以流的方式提供并来自于某些别的来源时。所以 HTTP 并不需要客户端必须在每个请求中发送 Content-Length 报头。不管怎样，HTTP 服务器有权利向任何给定的请求要求该报头值。
312 我们允许服务器中断任何在开始发送表述时没有提供 Content-Length 的请求，并且要求客户端重新提交具有 Content-Length 头部的请求。而该状态码将随着中断被发送回客户端。

如果客户端虚报了内容的长度，或者发送了过于庞大的表述，那么服务器将会中断该请求并关闭对应的连接，但是在该案例中，响应码应当是 413 (Request Entity Too Large)。

412 (Precondition Failed)

重要性：中等。

客户端会在它的请求报头中指定一个或多个先决条件，从而有效地告知服务器只有在特定条件满足时才执行它的请求。当这些条件并不满足时，服务器将不会执行该请求，取而代之是发送该状态码。

一个比较常见的先决条件是 `If-Unmodified-Since`（我在第 11 章中提到过）。客户端可以通过 `PUT` 请求来修改一个资源，但是要求只有当客户端最近一次获取该资源之后再也没有其他人对该资源进行过修改时，该修改才会生效。如果没有使用先决条件，客户端可能会在没有意识到的情况下覆盖其他人对资源的修改，或可能引起一个 409 (Conflict)。

响应报头：客户端可以通过使用 `If-Match`、`If-None-Match`、`If-Modified-Since` 或 `If-Unmodified-Since` 等中的任何一个报头来获取到该响应码。

`If-None-Match` 有一点特殊。如果客户端在发起 `GET` 或 `HEAD` 请求时指定了 `If-None-Match`，并且先决条件失败了，那么响应码将会是 304 (Not Modified) 而不是 412。这是有条件 HTTP `GET`（也在第 11 章中有所提及）的基础。如果在 `PUT`、`POST` 或 `DELETE` 请求中使用了 `If-None-Match`，并且先决条件判定失败了，那么响应码将是 412。当先决条件使用了 `If-Match` 或 `If-Unmodified-Since` 报头时，不管使用的 HTTP 方法是什么，响应码都将是 412。

413 (Request Entity Too Large)

重要性：低到中等。

与 411 (Length Required) 相似，服务器可以在中断客户端请求时使用该状态码，并在不等待请求完成的情况下关闭连接。411 状态码是针对那些没有指定它们表述长度的请求的，而该状态码则针对那些发送的表述过大而导致服务器无法处理的请求。

look-before-you-leap 请求（见第 11 章）是客户端用于避免被该错误中断的最好的方式。如果该 LBYL 请求获得了一个响应码为 100 (Continue) 的响应，那么客户端将可以继续提交整个表述。

响应报头：问题可能是暂时的，并且该问题是在服务器端（缺乏资源）而不是客户端的（表述过大）。如果是这样，服务器可以将 `Retry-After` 报头设置为一个日期或一个以秒为单位的数字，客户端可以在稍后进行重试。

313

414 (Request-URL Too Long)

重要性：低。

HTTP 标准没有指定关于 URL 长度的官方限制（在我看来，也不应该有任何的限制）。可是，大多数现有的 web 服务器都为 URL 的长度制定了一个上限，而 API 也都会这样做。而通常主要的原因是资源状态本应该在实体消息体中时，客户端却将它放入了 URL 中。而深度嵌套的数据结构也可以导致非常冗长的 URL。如果你遇到了该问题，请给你的资源分配一个由随机数字生成的不透明的 URL，而不是让 URL 的长度超过 1KB。

如果客户端连接到服务器并开始发送一个无限长的 URL，那么就算是一个没有预设最大 URL 长度的服务器最终也会采用 414 响应码来中断请求，从而释放 TCP 连接。服务器同样也可以选择简单的将连接终止掉。

415 (Unsupported Media Type)

重要性：中等。

服务器会在对客户端发送的表述所采用的媒体类型不理解时发送该状态码。服务器可能期望的是 `application/vnd.collection+json`，而客户端发送的却是 `application/json`。

如果客户端发送了一个拥有正确的媒体类型但是格式错误的文档（例如一个使用了错误的词汇表编写而成的 XML 文档，或者是一个使用了错误 ALPS profile 的 Collection+JSON 文档），更好的响应方式是选择更加通用的 400 (Bad Request)。

416 (Requested Range Not Satisfiable)

重要性：低。

当客户端要求的是表述中的一系列字节范围 (byte-ranges)，而表述实际上并没有这么大来满足这一字节范围时，服务器就会发送该状态码。换句话说，如果你向一个 99 字节的表述要求第 100 个字节，你将会得到该状态码。

请求报头：只有当原始请求包含了 Range 报头请求字段时，才会发送该状态码。如果原始请求包含了 If-Range 报头请求字段，该状态码将不会发送。

响应报头：服务器应该发送 Content-Range 字段来告诉客户端关于表述的确切大小。

417 (Expectation Failed)

重要性：低到中等。

该响应码是与 100 (Continue) 相对应的。如果你发起一个 look-before-you-leap 请求来查看服务器是否会接受你的表述，并且服务器表明它会接受，你将会得到响应码 100，那么你就可以继续了。如果服务器决定不接受你的表述，你得到响应码便会是 417，你将不应该再为发送你的表述而烦恼。

428 (Precondition Required)

重要性：中等。

定义方式：RFC 6585。

在第 11 章中，我建议的 API 实现需要客户端有条件地发起它们的 PUT 和 PATCH 请求，并将此作为一种规避丢失更新这一问题的方式。Web 服务器通过该状态码来实施规则，这就是说客户端的请求会因为没有带上条件化的报头而会被拒绝。

实体消息体：应该包含一个说明了服务器可以接受的条件化报头（很可能是 If-Match 或 If-Unmodified-Since）的文档。

429 (Too Many Requests)

重要性：中等。

定义方式：RFC 6585。

该状态码实施了服务器的请求限制策略。客户端近期发送了太多的请求，并且需要减少。

允许服务器在请求数违反了请求限制策略时简单地忽略掉请求，而不是向每个请求都做出 429 的响应。

响应报头：当服务器将再次接受来自客户端的请求时，Retry-After 报头应该会给出相应的提示。

实体消息体：应该包含一个用于说明请求限制策略的文档。

315

431 (Request Header Fields Too Large)

重要性：低。

定义方式：RFC 6585。

该响应码跟 413 (Request Entity Too Large) 或 414 (Request-URL Too Long) 很像，只不过这里的问题在于请求报头字段中的数据过多。

服务器对请求报头的大小做出预定义的限制是合法的，但是我并不认为这是一个好主意。特别是 Link 报头，它可以合乎情理地变得很大。如果客户端连接到服务器并发送了一个带有无限长报头的请求时，服务器可以通过发送 431 响应来中断该请求（服务器也可以简单地终止连接）。

实体消息体：如果一个特定的报头过大（而不是报头整体过大），实体消息体应该指明哪个报头是问题所在。

451 (Unavailable For Legal Reasons)

重要性：理想情况下非常低。

定义方式：互联网草案 “draft-tbray-http-legally-restricted-status”

客户端请求形式良好，但是服务器在合法的情况下需要拒绝它。通常是因为服务器通过某些审查制度禁止提供表述。服务器会同样在拒绝执行该资源状态转换时使用该状态码。

即使该请求是形式良好的，但这仍被认为是一种客户端错误，而合法性的要求存在于服务器。归根结底，该表述是因为某些原因而遭到审查的，你作为公民一定存在着什么问题。

5xx: Server-Side Error

5xx 系列的状态码用于表示服务器端的问题。这些状态码通常意味着服务器并不在一个可以执行客户端请求或甚至只是检查下请求是否正确状态，并且客户端应该在稍后重试它的请求。有时，服务器可以评估下客户端应该在何时重试它的请求，并将该信息放入到 Retry-After 响应报头中。

5xx 状态码比 4xx 状态码更少，并不是说服务器端会发生错误的事情就更少，而是因为过于详细并没有什么意义。客户端做不了任何事来修复服务器端的问题。

316 使用了这些状态码的响应将会在实体消息体中包括一份说明性的文档（或许是一个问题细节文档！）。

500 (Internal Server Error)

重要性：高。

这是一个通用的服务器错误响应。大部分的 web 框架会在它们运行的请求处理器代码抛出异常时发送该状态码。

501 (Not Implemented)

重要性：低。

客户端尝试使用服务器并不支持的 HTTP 特性（很可能是一项扩展特性）。

最通用的案例是当客户端尝试发起一个使用了某个类似 PATCH 这样的扩展 HTTP 方法，而某个普通的 web 服务器并不支持该扩展方法。这跟响应码 405 (Method Not Allowed) 非常相似，但是 405 暗示了客户端在某个不支持该方法的资源上使用了一种大家都认识的方法。501 响应码意味着服务器完全不认识该方法。

502 (Bad Gateway)

重要性：低。

你将只会从一个 HTTP 代理得到该响应码。它指示了问题存在于代理或在代理和上游服务器之间，而不是上游服务器。

如果代理完全无法连接到上游服务器，响应码将会是 504 (Gateway Timeout) 而不是 502 (Bad Gateway)。

503 (Service Unavailable)

重要性：中等到高。

该状态码意味着 HTTP 服务器已经启动，但是该 API 所基于的应用工作得并不正常。最可能的原因是资源紧缺：同时要求 API 处理的请求数过多。

因为导致该问题的原因很可能是因为重复的客户端请求，HTTP 服务器通常都可以选择拒绝接受某个客户端请求，而不是接受后仅发送一个 503 响应码。

响应报头：服务器可以发送一个 `Retry-After` 报头来告诉客户端什么时候可以再次提交请求。

504 (Gateway Timeout)

重要性：低。

跟 502 (Bad Gateway) 一样，你将只会从 HTTP 代理看到该状态码。该状态码表示代理无法连接到上游服务器。

505 (HTTP Version Not Supported)

重要性：非常低。

服务器并不支持客户端尝试使用的 HTTP 版本。直到 HTTP 2.0 发布前，你大概都不会看到该状态码。即使到了那时，你或许也只会在一台 HTTP 1.1 服务器上尝试使用 HTTP 2.0 的特性时才会看到该状态码。

实体消息体：应该会包含一个描述了当前服务器支持的 HTTP 版本的文档。

511 (Network Authentication Required)

重要性：中等。

定义方式：RFC 6585。

511 状态码是为了让 captive portals^{注1} 页面不那么烦人的一种尝试。captive portal 是当你尝试使用咖啡店或酒店的无线网络时接管你浏览器的网站。不管你请求的 web 页面是什么，captive portal 会提供你一个伴随状态码 200 (OK) 的页面来告诉你如何为你的网络访问付费。有时 portal 会提供你状态码 302 (Found)，并将你的浏览器重定向到一个告诉你如何为网络访问付费的页面。

当这些发生在你的 web 浏览器中时，这是非常让人厌烦的，但是你是一个人类。你可以阅读 web 页面并了解到如何合理地使用 Web。当这些发生在你的 API 客户端时，这就非常危险了。就自动化客户端而言，这看到去貌似是该 API 已经被关闭了，取而代之的一个难以辨认的，很可能在说“对不起，我们已经关闭了”的 HTML 文档。这将会导致客户端崩溃，并可能使得它的数据处于一个不一致的状态。

511 状态码并不能让一个 API 客户端能够更容易地通过 captive portal，但是它给了客户端一个机会来识别出发生了什么并可以优雅地退出，而不是恐慌和崩溃。

注1 一种让网络上的 HTTP 客户端在正常使用网络前强制访问某特殊网页（通常是基于认证的目的）的技术，常见于 wifi 热点的访问授权页面。——译者注

提供 511 状态码需要 captive portal 的开发者能在用户体验上多花点心思，所以我并不期待它能很快地被广泛使用起来。但是，如果当有人在咖啡店里启动你的 API 客户端时，在客户端收到 511 状态码能让你避免最糟糕的情况。

HTTP报头法典

HTTP 报头是一些用于描述 HTTP 请求或者响应的协议语义的元数据 (metadata)。一些报头, 比如 `If-None-Match` 只能用于请求中。客户端通过这种方式来告诉服务器如何处理该请求。有一些, 比如 `ETag` 只能用于响应中。服务器通过这种方式来向客户端传递一些信息, 这些信息或者是关于请求是如何被处理的, 或者是关于那些没有在表述中展示的相关的资源的。还有一些报头既可以用在请求中, 又可以用在响应中, 比如非常重要的 `Content-Type` 含有实体消息体的媒体类型。

对于标准的 HTTP 报头, 有两篇优秀的指导手册。一篇是 RFC2616, 即 HTTP 标准本身, 另一篇是已经出版的由 David Gourley 和 Brian Totty 编写的《HTTP: The Definitive Guide》(O'Reilly) 的附录 C。在本附录中, 我会对标准 HTTP 报头给予一些简单描述, 这些描述着眼于它们在 RESTful API 中的用法, 而不是它们在网站以及 HTTP 代理等其他基于 HTTP 应用的用法。

自定义HTTP报头

创建一个新的 HTTP 方法或者状态码的代价是相当大的。最起码, 这需要编写一个 RFC。但是任何运行 HTTP 服务器的人都可以定义他们自己的 HTTP 报头。AtomPub 定义了一个叫作 `Slug` (将在后面章节进行介绍) 的 HTTP 报头。Amazon 也为 S3 API 定义了像 `X-amz-acl` 和 `X-amz-date` 这样的报头。

在《RESTful Web Services》一书中, 我为何时定义一个自定义的 HTTP 报头和怎样为这个报头命名给出了一些建议。在过去几年里, 我改变了对这两件事情的观点。你很可能一点儿也不应该创建一个新的 HTTP 报头。

一个新的 HTTP 报头就像新的 HTTP 方法和状态码一样，是一个 HTTP 扩展。如果你创建了一个新的报头，你必须像已有的标准 HTTP 报头那样：用大量措辞详尽的人类可读的资料将这个新报头记录下来。它们之间的区别就是当你的用户在学习了你自定义的 HTTP 报头之后，他们并不能将这些知识应用到其他的 HTTP 服务器上。你自定义的报头是一种特定于你的 API 的 fiat 标准。

好消息就是许多使用新 HTTP 报头的用例已经不再适用了。和几年前相比，超媒体格式现在已经非常多而且更加灵活。过去写在自定义报头中的信息现在可以直接写到一个表述中了。如果你需要为现有的媒体类型添加新的应用语义，你可以将这些信息记录到机器可读的 profile 中，来代替创建新的 HTTP 报头。

你应该只有在发现 HTTP 协议本身缺少某些内容时才去创建一个新的 HTTP 报头。如果你的报头成为了 HTTP 的标准扩展，这是否是最好的归宿？在这种情况下，你可以放手去定义一个新的报头。可能将来有一天它会变成一个像 Link 报头那样的标准扩展。

至于命名：我过去建议自定义的 header 的名称要以字符串 X- 开头来表示这是一个“扩展（extension）”。假设你想要将你自定义的报头称为 My-Header，我之前建议你将它用 X-My-Header 来代替。RFC6648 改变了我的观点。你不应该这么做。你应该顺其自然地使用 My-Header。

来自其他协议的总结出来的经验教训（见 RFC 6646）告诉我们 X- 前缀给我们带来的麻烦要远远多于它的贡献。如果你自定义的 HTTP 报头被标准化，那么你将不能够移除 X-，因为这会破坏所有已有的客户端。这意味着 X- 前缀并不确实表示扩展（extension）的存在。因为 X- 不再起到它当初所设计的作用，所以我们也就不必要使用它了。你应该选择一个准确并且唯一的名称来开始用于自定义的报头。

报头

RFC2616 记录了 46 个报头，同时，扩展 RFC(extension RFC) 以及一些互联网草案另外定义了 8 个报头。我将对每个报头用一个小篇幅内容来进行介绍，我会说明它们是否出现在 HTTP 请求、响应或者两者兼有。我会针对它们在 API 中的重要性给出个人意见。我还会通过一段简短的描述来说明这些报头，对于难以理解或者特别重要的报头，篇幅会更长一些。我不会详细到说明每个报头的值的类型、可选项等细节内容。我认为你足够聪明并能够在需要的时候查找更详细的信息。

除非另外说明，一个报头的正式定义可以在 RFC 2616 中找到。

Accept

类型：请求报头。

重要程度：中等。

客户端会通过发送一个 `Accept` 报头来告诉服务器，它更愿意服务器采用哪种媒体类型作为自己的表述。这是我在第 11 章中介绍过的“内容协商”技术。有的客户端可能想要一份 XML 格式的 HAL 文档 (`Accept: application/hal+xml`)，有的客户端想要的可能是同一份 HAL 文档的 HAL+JSON 表述 (`Accept: application/hal+json`)。

如果你想要为这个报头（或者任何其他 `Accept-*` 类型的报头）实现一个解释器，可以参考 RFC 2616 来了解更细节的内容。这个格式要比你所想象的要复杂得多。

Accept-Charset

类型：请求报头。

重要等级：低。

客户端通过发送一个 `Accept-Charset` 报头来告诉服务器，它希望服务器在它的表述中采用什么字符编码。有的客户端可能希望将某个含有日文的资源的表述以 UTF-8 编码，有的客户端可能希望采用 Shift-JIS 来对同样的数据进行编码。

就我个人而言，我认为大家都应该使用 UTF-8 或者 UTF-16 来对所有内容进行编码。

Accept-Encoding

类型：请求报头。

重要程度：中等到高。

客户端通过发送 `Accept-Encoding` 报头来告诉服务器，它可以采用 `gzip` 等一些众所周知的算法对响应的实体消息体进行压缩以节省带宽。尽管名称中有 `encoding`，但是它和字符编码没有任何关系。字符编码是由 `Accept-Charset` 负责的。

`Accept-Encoding` 的值被称为“内容编码 (content-coding)”。IANA 在 <http://www.iana.org/assignments/http-parameters/http-parameters.xml> 维护了一份用于记录合格的内容编码的注册表。通常，内容编码只用于在网络上对数据进行压缩。

Accept-Language

类型：请求报头。

重要程度：低。

322 客户端可以通过发送 **Accept-Language** 报头来告诉服务器，它希望服务器在表述中采用哪种人类语言。这并不影响表述的格式，但是，它会影响到数据。

Accept-Language 的值被称为语言标签 (language tag)。你很可能熟悉一些语言标签：英语是 **en**，具体到美式英语则是 **en-us**。RFC5646 为语言标签制定了格式，而 IANA 也在它的网页上以机器可读的形式维护了一份语言 (**en**) 和地区 (**us**) 的注册表。

Accept-Ranges

类型：响应报头。

重要程度：低到中等。

服务器通过发送本报头来表明它支持对某个资源执行部分 HTTP GET (partial HTTP GET) 操作 (见第 11 章)。该报头的值必须是字符串 “bytes”。

Accept-Ranges: bytes

它应该只有在客户端下载某个表述的过程被中断后才能出现。如果服务器在最初的响应中就设置了 **Accept-Ranges** 报头，那么客户端就能知道它可以通过提供合适的 **Range** 报头向同一个 URL 上发起第二次请求。客户端然后就可以在上次中断的地方重新启动下载过程，而不需要再下载整个表述。

Age

类型：响应报头。

重要程度：低。

如果响应的实体消息体不是服务器刚刚生成的，那么 **Age** 报头可以用于以秒为单位来衡量该实体消息体在服务器上已经存在的时间。该报头通常是由 HTTP 缓存来设置的，这样客户端就意识到它可能收到的是一个旧的表述副本。

Allow

类型：响应报头。

重要程度：低到中等。

我在第 3 章中简要讨论过这个报头。它是作为 OPTIONS 请求的响应而被发送出去的，它会将资源的一些协议语义告诉客户端，具体来说就是这个资源会响应哪些 HTTP 方法。

这个报头并不是非常重要，因为超媒体提供了一种比 OPTIONS 方法更好的发现机制。但是有些 API 需要实现该报头来支持 OPTIONS 请求。

323

Authorization

类型：请求报头。

重要等级：非常高。

这个请求报头包含了授权证书，比如客户端按照某些商定的方案编码过用户名和密码。服务器会对这些证书解码并决定是否执行相应的请求。

这应该是你唯一需要的认证报头（除了 Proxy-Authorization，它是工作在另外一个层面上的），因为它是可扩展的。最常见的方案是 OAuth 和 HTTP Basic，但是只要客户端和服务器端都能够理解，任何方案都是可以的。

还有一些其他的基于 Authentication 而作用的认证报头，尤其是 X-WSSE，但是这些标准都几乎已经消失殆尽了，所以我也就不在本书中进行介绍。

Cache-Control

类型：请求和响应报头。

重要等级：高。

这个报头包含了一个发给客户端以及服务器端的缓存（包括客户端的本地缓存以及服务器器本身的缓存）的指令。它明确了应该如何对数据进行缓存以及何时从缓存中删除数据的规则。这是一个非常复杂的报头，但是我在第 11 章中介绍了最基本的缓存指令（“cache”和“don't cache”）。

Connection

类型：响应报头。

重要等级：低。

一个 HTTP 响应的大部分内容是服务器端发给客户端的一则消息。代理等中间媒介能够

查看响应，但是这其中并没有任何针对它们的内容。但是服务器可以额外插入一些针对代理的报头，代理又可以针对链路中的下一个代理插入其他报头。当这种情况发生时，这些特殊的 header 的名称被添加在 **Connection** 报头中。这些报头会作用到机器与机器之间的 TCP 连接，而不是服务器和客户端之间的 HTTP 链接。在把响应传递出去之前，代理应该先去掉那些特殊的报头和 **Connection** 报头本身。当然，如果它希望的话，它可以添加自己特殊的用于通信的报头以及新的 **Connection** 报头。

如下是一个简单示例，当然，这和本书的关系并不紧密。服务器可以在响应中发送如下 3 个 HTTP 报头，这个响应会通过一个代理：

```
Content-Type: text/plain
Proxy-Directive: Deliver this as fast as you can!
Connection: Proxy-Directive
```

Proxy-Directive 是一个自定义的 HTTP 报头。服务器和代理都能理解它的含义，但是客户端或许不能。代理会去除 **Proxy-Directive** 和 **Connection** 这两个报头，然后将仅剩的一个报头发送给客户端：

```
Content-Type: text/plain
```

如果你正在编写一个客户端而且没有使用代理服务器，那么你可能会看到 **Connection** 的值是 **close**。这仅仅是说服务器在完成这个请求后会关闭 TCP 连接。

Content-Disposition

类型：响应报头。

重要程度：中等。

出处：RFC 6266。

Content-Disposition 报头通常用于表明客户端应该将实体消息体保存为一个文件，而不是将其作为一个表述进行处理。当使用时，样式如下：

```
Content-Disposition: attachment; filename="bug-1234-attachment-1"
```

任何能够保存上传的文件的 API 都应该使用 **Content-Disposition** 来将上传的文件与 API 生成的文档区别开来。如果 API 提供了一个很怪异的文档，这意味着这个 API 有 bug——除非这个文档是由客户端上传的，并且 API 仅仅是准确地将所上传的内容展示出来。

在上面这个例子中，值 **attachment** 说明这个实体消息体是一个附件（**attachment**），而参数 **filename** 表明要以什么文件名来进行保存。这个参数会导致安全方面更加复杂，这

也是 Content-Disposition 报头被明确地排除在 HTTP 标准之外的原因。当编写涉及 Content-Disposition 的客户端，或者当允许 API 客户端对它们上传的文件命名时，请查看 RFC 6266 的建议，请一定要小心谨慎。

Content-Encoding

类型：响应报头。

重要等级：中等到高。

这个响应报头是与请求报头 Accept-Encoding 所对应的报头。请求报头会要求服务器用某种算法来对实体消息体进行压缩。而这个报头就告诉客户端，如果有，服务器到底采用了哪种算法。

就跟 Accept-Encoding 一样，这个报头的值也被称为“内容编码”，并且 IANA 在 <http://www.iana.org/assignments/http-parameters/http-parameters.xml> 上维护了一份合格的内容编码注册表。从理论上说，内容编码可以是任何种类的可逆数据变换，但是所有注册的内容编码都是数据压缩格式。

Content-Language

类型：响应报头。

重要程度：中等。

这个响应报头是与请求报头 Accept-Language 所对应的报头，或者它与资源的 URI 中的一套变量集合所对应，它声明了人类必须懂得的用以了解实体消息体的含义的自然语言。

就像所有的 Accept-* 报头以及它们对应的响应报头一样，这个报头可以拥有多个值。如果某个实体消息体是一部配有日文字幕的中文普通话电影，它的 Content-Language 可能是 zh-guoyu, jp。如果电影中只出现了一句英文短语，en 应该不会出现在 Content-Language 报头中。

Content-Length

类型：响应报头。

重要程度：高。

这个响应报头表示实体消息体的以字节为单位的大小。有两个原因决定了这个报头很

重要：首先，客户端能够提前读取这个报头并根据实体消息体的大小来做好准备。第二，客户端可以发起一个 HEAD 请求来明确实体消息体的大小，而不用真的请求它。Content-Length 的值会影响客户端的决策，是直接获取整个实体消息体还是使用 Range 报头来获取一部分内容，还是根本不获取。

Content-Location

类型：响应报头。

326

重要程度：低。

这个报头会告诉客户端它所请求的资源的标准 URL。不同于 Location 报头的值，它纯粹是说明性的信息。客户端并不被期望要开始使用这个新的 URL。

它主要用于那些为单个资源的不同表述分配不同的 URL 的 API。如果客户端想要链接到通过内容协商而获取到的特定表述，它可以使用 Content-Location 提供的 URI。这样，如果你请求 `/documents/104`，并且使用 Accept 和 Accept-Language 报头来指定用英语记录的 HTML 表述，你可能收到一个指定 `/documents/104.html.en` 为 Content-Location 报头的值的表述。这是一个链接到资源的特定表述的链接。

注意：这个报头是一个简单的超媒体控件。它的作用和采用 IANA 注册链接关系 canonical 的链接是一样的。

Content-MD5

类型：响应报头。

重要程度：低到中等。

它是实体消息体的加密校验码。客户端能够使用它来检查实体消息体是否在传输过程中被损坏。攻击者（比如客户端和服务端数据传输过程中的中间人）能够同时篡改实体消息体以及 Content-MD5 报头，所以它对安全性并没什么好处，它仅仅是对错误检测有好处。

Content-Range

类型：响应报头。

重要程度：低到中等。

当客户端使用 Range 请求报头发起一个部分 GET 请求（partial GET 报头）时，这个响

应报头会说明客户端正在接收哪一部分表述。

Content-Type

类型：请求和响应报头。

重要程度：非常高。

它是最出名的 HTTP 报头，很可能也是最重要的。Content-Type 提供了实体消息体的媒体类型。媒体类型有以下 3 个用途：

- 它决定了接收者应该使用哪种解析器来对实体消息体进行解析。
- 它经常决定了表述的协议语义——哪部分表述是超媒体控件、激活那些控件会触发哪些 HTTP 请求。
- 它也同样决定了表述的应用语义——该表述在这套特定的 API 以及在现实世界的概念中的含义。

327

虽然有一些其他的方式来传递应用和协议语义，比如链接到 profile 的链接，但是 Content-Type 是主要方式。将 application/json 作为你的媒体类型是一个糟糕的主意，这就是原因。你这是在拒绝一个非常大的机会。

当提供的文档是由一种媒体类型和一个 profile 进行描述时，你应该提供一个 Link 报头来链接到这个 profile。

Cookie

类型：请求报头。

重要程度：在人类 web 上高，在 API 世界中低。

出处：RFC2109。

这很可能是仅次于 Content-Type 而最著名的 HTTP 报头了。但是，它并没有出现在 HTTP 标准中；它是一个 Netscape 扩展。

cookie 是客户端和服务端之间达成的一种约定。按照这个约定，服务器通过使用 Set-Cookie 报头（后面章节中将对此展开介绍）来在客户端保存一些半持久化的状态信息。一旦客户端得到一个 cookie，它就应该在随后的所有 HTTP 请求中通过把所有 cookie 都设置到 Cookie 报头中的方式将这个 cookie 传递回去。因为这些数据在无形中都提交到每次请求的 HTTP 报头中，这看起来像是客户端和服务端在共享状态。

Cookies 在 REST 圈子中的名声非常差,这其中有两个原因。第一,它们所包含的“状态”常常只是一个 session ID:一个很短的由数字和字母组成的字符串键,它和服务端端的某个非常大的数据结构相关联。它破坏了无状态原则,因为应用状态现在被保存在服务器上了。

更进一步,一旦客户端接受了一个 cookie,它就应该与随后一定时间内的所有请求一起提交给服务器。服务器也会要求客户端以后不能再发起在接受这个 cookie 之前曾经发起的请求了。这也违反了无状态原则。

如果你必须使用 cookie,请保证你将所有的状态都保存在客户端。否则,你会丢掉许多 REST 的可伸缩性的优势。

Date

类型:请求和响应报头。

328 ▶ 重要程度:作为请求报头高,作为响应报头低。

作为请求报头,它表示请求发起时客户端上的时间。当作为响应报头时,它表示的是请求被处理完成后服务器端的时间。在计算某个缓存的文档是否还是新的时,缓存会使用响应报头的 Date。

ETag

类型:响应报头。

重要程度:非常高。

ETag 的值是一个和某个特定版本的表述相关联的字符串。任何时候,只要表述发生了变化,ETag 也应该改变。

服务器应该尽可能在 GET 请求的响应中发送 ETag。正如我在第 11 章所展示的那样,客户端可以通过将之前的 ETag 的值作为 If-None-Match 请求报头的值来发送,进而发起一次条件 GET 请求 (conditional GET request)。如果表述没有发生改变,那么 ETag 也就没有变化,这样服务器就可以不用发送表述来节省时间和带宽。

条件 GET 请求的主要推动者是更加简单的响应报头 Last-Modified 以及对应的请求报头 If-Modified-Since。ETag 的主要用途是提供第二道防线。如果表述在一秒内改变了两次,它在 Last-Modified-Since 中都仅仅呈现一个值,但是 ETag 却是两个不同的值。

如果在你的服务器和客户端之间有中间媒介修改了你的表述 (比如 Apache 的 mod_

compress 模块，它会透明地对表述进行压缩)，这个中间媒介也将修改 **ETag** 的值，而这可能会条件请求（conditional request）失效。

Expect

类型：请求报头。

重要程度：中等，但是很少被使用。

这个报头用于表示 look-before-you-leap 请求（见第 11 章）。如果客户端应该行动（“leap”）起来并发起真正的请求，服务器会发送响应码 100（Continue），如果客户端不应该行动，服务器会发送响应码 417（Expectation Failed）。

Expires

类型：响应报头。

重要程度：中等。

329

这个报头告诉客户端或者位于服务器与客户端之间的代理，它可以缓存 HTTP 响应（而不仅仅是实体消息体！）到某一时间。这非常有用，因为即使是那些最终什么也不需要做的条件请求也需要发起 HTTP 请求的开销。而通过使用 Expires，客户端可以免去发送任何 HTTP 请求的需要——至少在一定时间内。

通常使用 Cache-Control 报头的 max-age 缓存指令（见第 11 章）会更容易一些。这是指，相比于要计算一个小时后的准确时间，它更易于说明“这个表述在一小时内应该是正确的”。

但是如果服务器准确知道表述什么时候会改变（因为它会在每星期、每天、每小时的相同时间发生改变），Expires 就更合适一些。

客户端应该将 Expires 的值作为一个粗略的指导，而不应认为这是一种约定，约定实体消息体直到那个时候才会改变。

From

类型：请求报头。

重要程度：非常低。

这个报头的作用类似于 email 消息中的 From 报头。它提供了发起这个请求的人的邮件

地址。由于隐私问题它从来没有在万维网中被使用过，并且它也从来没有在 web API 的世界中使用过，因为我们有类似于 OAuth 令牌的方式来识别客户端。

Host

类型：请求报头。

重要程度：必需。

这个报头包含了请求 URL 中的域名部分。如果客户端对 `http://www.example.com/page.html` 发起了一个 GET 请求，那么真正被请求的 URL 是 `/page.html`，而 Host 报头的值是 `www.example.com` 或者 `www.example.com:80`。

从客户端的角度看，它作为一个必须设置的报头可能看起来很奇怪。它之所以是必须设置的，这是因为 HTTP 服务器可以在单个 IP 地址上支持任意数量的域名。这个功能被称为“基于名称的虚拟主机 (name-based virtual hosting)”，并且它可以使拥有多个域名的人免于必须为每个域名购买单独的计算机和 / 或网络。

基于名称的虚拟主机的问题就是，当客户端打开一个 TCP 连接时，它是通过 IP 地址而不是域名连接到服务器上的。Host 报头不包含域名的话，HTTP 服务器就不能知道客户端所请求的目标是哪一台虚拟主机了。

330

If-Match

类型：请求报头。

重要程度：高。

这个报头最好用其他报头来进行描述。它像 If-Unmodified-Since 那样被用于除 GET 条件 (conditional) 请求之外的 HTTP 操作请求——通常用于避免我在第 11 章中提到的更新丢失的问题。但是 If-Unmodified-Since 采用某个时间作为它的值，而这个报头采用一个 ETag 作为它的值。

简单说，这个报头同 If-None-Match 和 ETag 的关系就相当于 If-Unmodified-Since 同 If-Modified-Since 和 Last-Modified 的关系那样。

If-Modified-Since

类型：请求报头。

重要程度：非常高。

这个请求报头支撑着条件 HTTP GET(conditional HTTP GET)。它的值就是客户端在上一次向某个资源发起的 GET 请求后收到的响应报头的 Last-Modified 的值。当客户端将这个值作为 If-Modified-Since 的值发送以后，它是在请求：只有当表述自上次请求后发生了改变时才获取这个表述。

如果表述在上次请求后实际上发生了改变，那么它的新 Last-Modified 日期就比之前的那个日期还要新。这就意味着 If-Modified-Since 的条件已经满足，服务器就会发送新的表述出去。如果该资源没有变化，Last-Modified 日期就和之前的一样，那么 If-Modified-Since 条件就没有满足。服务器会发送响应码 304 (Not Modified)，并且不发送任何实体消息体。这就是，如果这个条件不满足，条件 HTTP GET 就会成功。

因为 Last-Modified 的精度只是一秒，如果条件 HTTP GET 只依赖于 If-Modified-Since，那么它可能会偶尔提供错误的结果。这也就是我们同时使用 ETag 和 If-None-Match 的原因。

If-None-Match

类型：请求报头。

重要程度：非常高。

这个报头同样用于条件 HTTP GET。它的值来自于通过上一次 GET 请求而得到的 ETag 响应报头。

331

如果表述的 ETag 在上次请求之后发生了改变，那么 If-None-Match 条件就成功了，服务器会发送新的表述。如果该 ETag 和之前一样，那么这个条件就失败了，服务器会发送响应码 304 (“Not Modified”)，并且不发送实体消息体。

If-Range

类型：请求报头。

重要程度：低。

这个报头用于发起一个条件部分 GET 请求 (conditional partial GET request)。它的值来自于前一个 range 请求所收到的 ETag 或者 Last-Modified 响应报头。服务器只在客户端所请求的那部分实体消息体的发生改变时才会发送新的那部分表述。否则服务器会发送 304 (Not Modified)，即便实体消息体的其他部分的内容发生了改变。

条件部分 GET 请求不经常被使用到，因为客户端不大可能只获取一个大的表述的一部分

数据，然后试图在以后只获取这部分同样的数据。

If-Unmodified-Since

类型：请求报头。

重要程度：中等。

正常情况下，客户端使用响应报头 **Last-Modified** 的值作为请求报头 **If-Modified-Since** 的值来实现条件 GET 请求。这个报头也会采用 **Last-Modified** 的值，但是它通常用于发起除 GET 请求外的条件请求。它的目标通常是避免我在第 11 章中提到的更新丢失问题。

如果你发起带有 **If-Unmodified-Since** 条件的 PUT 或者 PATCH 请求，那么，如果其他人在你不知情的情况下修改了这个资源，你的请求会一直收到状态码 412 (**Precondition Failed**)。你可以重新获取这个表述并决定如何处理这个其他人修改过的新版本的表述。

这个报头同样可以用于 GET。请参见 **Range** 报头条目中的例子。

Last-Modified

类型：响应报头。

332 重要程度：非常高。

这个报头使得条件 HTTP GET 成为可能。它会告诉客户端表述上一次修改的时间。客户端可以记录下这个日期并将它用到未来请求的 **If-Modified-Since** 报头中。

在 web 应用中，**Last-Modified** 通常是当前时间，这就使得条件 HTTP GET 没什么用处了。API 应该做的更好，因为 API 客户端经常会向服务器的同一个 URL（尤其是告示牌 URL）一次次频繁地发起请求。

Link

类型：请求和响应报头。

出处：RFC5988

这个报头是作为一个通用的超媒体链接而提供的，我在本书的第 4 章和第 11 章中对这个报头已经介绍过很多次了。它的值包括一个由括号括起来的 URL，以及一些用来提供这个 URL 的上下文的参数（比如 **rel**）。比如：

```
Link: <http://www.example.com/story/part2>; rel="next"
```

虽然 `rel` 是这个报头最重要的参数，但是 RFC5988 也定义了一些其他的参数：`hreflang`、`media`、`title`、`title*` 和 `type`。`hreflang` 和 `type` 参数就像它们在 HTML 的 `<a>` 标签的作用一样，分别指定了目标链接的人类语言和媒体类型。而 `title` 和 `title*` 参数分别以不同的方式为这个链接提供了人类可读的标题。

`media` 参数的作用和 HTML 的 `<style>` 标签的 `media` 属性是相同的：它说明了展示链接另一端的表述所采用的显示媒介（`screen`、`print`、`braille` 等）。

虽然 Link 通常是作为一个响应报头，但是在某些情况下，客户端需要将 Link 报头和请求一起发送。如果有一份文档需要通过某个 profile 来理解它的应用语义，那么客户端在通过 POST、PUT、PATCH 请求来操作这个文档时，应当将该文档和 Link 报头一起发送给服务器。Link 的值应该指向那个 profile 文档，并附加 `rel="profile"` 信息。这一逻辑对于一个需要使用 JSON-LD 上下文才能理解的文档是同样有效的。

为 Link 报头提供多个值是合法的：

```
Link: </story/part3>; rel="next", </story/part1>; rel="previous"
```

或者一次发送多个 Link 报头：

```
Link: </story/part3>; rel="next"  
Link: </story/part1>; rel="previous"
```

这对于一些其他的 HTTP 报头是同样适用的，不过 Link 和 Link-Template 可能是你最需要使用这一功能的报头。 333

Link-Template

类型：响应报头。

出处：失效的互联网草案“draft-nottingham-link-template”。

我在第 11 章中对此报头进行过介绍。除了它的值是一个 URI 模板（RFC5988）以外，它的作用类似于 Link。它提供了和 `action="GET"` 的 HTML 表单差不多的超媒体的能力。它比 Link 要灵活得多，Link 报头也就和 HTML 的 `<a>` 标签差不多。

Link-Template 报头支持所有的 Link 报头的参数，外加一个我在第 11 章介绍过的“var-base”参数。

Location

类型：响应报头。

重要程度：非常高。

这是 RFC2616 所定义的两个担当超媒体链接的 HTTP 报头中的一个。另外一个报头是 Content-Location，它的意思也很简单并始终一致。但是 Location 链接的含义取决于状态码。

这个报头同 3xx (Redirection) 系列的状态码有紧密联系，对每种类型的重定向而言，Location 的含义都稍微有些不同，许多围绕 HTTP 重定向的困惑都同这一事实相关。

- 当客户端的请求创建了一个全新的资源时，响应码是 201 (Created)，而 Location 报头就是链接到这个新创建的资源链接。
- 当服务器不能决定提供哪种表述并且每个表述都有自己的 URL 时，状态码是 300 (+Multiple Choices+)，并且 Location 报头链接到服务器所喜欢的表述：它不同于一种更常见的情况：服务器有多个表述，每个表述都有自己的 URL，客户端通过内容协商来在它们之中进行选择。在那种情况下，状态码是 200(+OK+)，不需要提供 Location，而 Content-Location 指向客户端所商定的表述的标准 URL。
- 当客户端的请求导致资源改变了自己的 URL 时，响应码是 301(Moved Permanently)，而 Location 报头则链接到这个原始资源的新位置。
- 当客户端向一个“错误的”URL 发起了一个请求，但是服务器仍然可以理解客户端所想要引用的资源时，Location 报头会链接到“正确的”URL。响应码可以是 301、302 (Found)、307 (Temporary Redirect) 或者 308 (Permanent Redirect)，这取决于这个 URL 是“错误的”URL 的确切原因。
- 当响应码是 303 (See Other) 时，Location 另一端的资源不是所请求的资源的表述，而是一条消息，这条消息用于说明这个请求是如何被处理的。它通常供那些能响应 POST 请求但是本身并不拥有表述的资源使用。

Max-Forwards

类型：请求报头。

重要程度：非常低。

这个报头主要用于 TRACE 方法，TRACE 方法是用于跟踪那些处理客户端的 HTTP 请求的代理。我并没有在本书中对 TRACE 展开介绍，但是作为 TRACE 请求的一部分，Max-Forwards 用于限制一个请求可以经过的代理的数目。

Pragma

类型：请求或响应报头。

重要程度：非常低。

Pragma 是一个用于在客户端、服务器以及中间媒介（如代理）之间传递特殊指令的报头。唯一正式的 pragma 值是 no-cache，它在 HTTP1.1 中已经被废弃了；它和将 Cache-Control 报头设置为 no-cache 的作用是相同的。

你可以定义你自己的 HTTP pragma，但是更好的办法是定义你自己的 HTTP 报头（并不是说你应该从二者中选择一个做）。

Prefer

类型：请求报头。

重要程度：当前低，将来可能高。

出处：互联网草案“draft-snell-http-prefer”。

这个 Prefer 报头允许客户端就各种各样的小问题将它的首选项配置（preference）发送给服务器，这些问题都是没有在 HTTP 标准或者某种媒体类型相关的规则中提到过的。互联网草案“draft-snell-http-prefer”定义了 Prefer 报头并提出一份其他 web 标准可以补充的 IANA 首选项注册表，但是它也定义了 6 种首选项用于处理 3 种经常在 API 中出现的问题：

◀ 335

- **handling=lenient** 首选项会告诉服务器，即便请求中有一些语法或者语义的小问题，它也要尽量来处理它。**handling=strict** 首选项正好相反：它是告诉服务器，只要它发现任何细微的问题都要返回错误状态。
- **respond-async** 首选项会告诉服务器，如果处理完一个请求要花费很长的时间，客户端想要服务器发送响应码 202 (Accepted)，而非要客户端来等待那个表示请求处理完成的响应。**wait** 首选项可以设置为若干秒（比如 **wait=10**），用来表示客户端愿意等待多久来获取一个真正的响应。
- **return=minimal** 首选项是和那些创建或者修改某个资源的请求（PUT、POST 或 PATCH）一起发送出去的。客户端通过这种方式来告诉服务器，它不想要这个新创建或者修改过的资源的完整的表述。**return=representation** 首选项正相反。它的意思是客户端想要获取完整的表述，即便正常情况下服务器并不会发送该表述。

如果你定义了你自己的首选项，你要记住它们拥有和自定义 HTTP 报头一样的问题。一

个新的首选项就是一个 HTTP 的扩展，你必须像其他首选项所做的那样非常详细地将它记录下来。你新的首选项是一个 fiat 标准，所以大多数客户端不会支持它。总而言之，通常，创建一种新的超媒体控件比定义一个新的首选项更容易。

Preference-Applied

类型：响应报头。

重要程度：仅次于 Prefer。

出处：互联网草案 “draft-snell-http-prefer”。

当服务器收到一个使用了 Prefer 报头的请求，并且决定配合客户端的某些首选项时，它会在 Preference-Applied 报头中提到它接受了哪些首选项。有时候，一个错误的出现是由于 handling=strict 还是由于的确发生了错误，或者实体消息体非常小是由于 return=minimal 还是由于表述确实是小的，这些并不清楚。而有了 Preference-Applied 报头，一切都可以真相大白。

336

Proxy-Authenticate

类型：响应报头。

重要程度：低到中等。

一些客户端（特别是在公司环境中）只能够通过代理服务器来获得 HTTP 访问权限。有一些代理服务器需要认证。代理服务器就是通过这个报头来要求认证的。它会随着响应码 407 (Proxy Authentication Required) 一起发送过去，它的作用和 WWW-Authenticate 是类似的，只不过它是告诉客户端如何通过代理服务器的认证而非通过 web 服务器的认证。

WWW-Authenticate 与 Authorization 相对应，而 Proxy-Authenticate 与 Proxy-Authorization（见下一节）相对应。单个的请求可能需要同时包括 Authorization 和 Proxy-Authorization 两个报头：一个用来认证 API，另一个用来认证代理服务器。

由于大部分 API 在它们架构中并不包含可见的代理，所以这个报头并不是和本书特别相关的主题。但是如果在客户端和外部的 Web 之间存在代理服务器，这个报头可能就和客户端相关了。

Proxy-Authorization

类型：请求报头。

重要程度：低到中等。

这个报头用于让一个请求通过代理服务器的认证。它的作用类似于 `Authorization`。它的格式取决于 `Proxy-Authenticate` 所定义的方案，就像 `Authorization` 的格式取决于 `WWW-Authenticate` 所定义的方案那样。

Range

类型：请求报头。

重要程度：中等。

这个报头表示，客户端试图只请求资源的部分表述（见第 11 章）。客户端通常会由于在之前试图下载一个比较大的表述时被中断而发送这个报头。这样，它就能继续下载剩下的那部分表述了。正是因为如此，这个报头通常和 `Unless-Modified-Since` 相结合。如果表述在你上次请求之后发生了变化，你就需要从头开始获取整个表述。

Referer

337

类型：请求报头。

重要程度：对于万维网而言高，对于 API 而言低。

当你单击你的 web 浏览器上的一个链接时，浏览器发送的 HTTP 请求中会有一个 `Referer` 报头，它的值是你当前所在的网页的 URL。就是这个 URL 把你的客户端“引荐(refered)”给你正在请求的 URI。是的，报头的拼写的确是不标准的（正确的拼写是 `referrer`）。

尽管这个报头在人类 web 中很常见，但是它很少用于 API 中。它可以用于传递部分应用状态（客户端执行 API 过程中的上一步的路线）给服务器。

尽管 `Referer` 的值总是一个 URL，但是我不认为它是一个超媒体链接，这是因为这个报头是客户端发送给服务器的。超媒体链接是从服务器端发送给客户端的。

Retry-After

类型：响应报头。

重要程度：低到中等。

这个报头通常和表示失败的响应码一起出现：或者是 413 (`Request Entity Too Large`)，或者是 429 (`Too Many Requests`)，还有可能是 5xx (`Server-side error`)

系列状态码中的某一个。这些代码是告诉客户端，服务器现在不能处理该请求，但是或许在稍后的时间里，它能够处理同样的请求。**Retry-After** 报头的值是客户端可以再次尝试的时间或者客户端应该等待的秒数。

如果问题是由于服务器过载，而服务器使用相同的规则来设定每个客户端的 **Retry-After** 的值，这仅仅是使这些相同的客户端晚些时候以相同的顺序发起同样的请求，服务器会再次过载。服务器应该使用一些随机技术来修改 **Retry-After**，就像处理以太网的退避周期（backoff period）那样。

Set-Cookie

类型：响应报头。

重要程度：对于万维网而言高，对于 API 而言低。

出处：RFC 2106。

服务器通过该报头将一些半持久化状态设置到客户端的 cookie 中。客户端应该在随后所有的请求中一直发送适当的 **Cookie** 报头，直至到达这个 cookie 的过期时间（expiration date）。客户端可以忽略这个报头（这通常是一个好主意），但是并不能保证它们在不提供 **Cookie** 报头的情况下后面发起的请求会收到正确的响应。它违反了无状态行原则，这就是我不建议在 API 中使用 cookie 的原因。

338

Slug

类型：请求报头。

重要程度：相当高，但是只出现于 AtomPub API。

出处：RFC 5023。

当一个 AtomPub 客户端 POST 一个二进制文档（比如一幅图片）给某个 feed 时，它可以在 **Slug** 报头中为这个文档添加一个标题。这使得上传只需要一步，而不再需要经过两个步骤（使用 POST 来上传文件，使用 PUT 来编辑它的元数据（metadata））。

TE

类型：请求报头。

重要程度：低。

这是另一个 **Accept** 类型的报头，它让客户端声明它所能接受的传输编码（transfer encoding）（见 **Transfer-Encoding** 一节对传输编码的介绍）。《HTTP: The Definitive Guide》一书指出，采用 **Accept-Transfer-Encoding** 的名称也许更合适。

TE 的值被称为“传输码”，IANA 保存了一份合格的传输码注册表。

Trailer

类型：响应报头。

重要程度：低。

当服务器使用 **chunked** 传输编码来发送实体消息体时，它可以选择将某些 HTTP 报头记录到实体消息体的后面而非前面（详细内容见后面）。这就使得它们从报头变成了 trailer。当服务器想要将某个报头作为 trailer 发送时，服务器便会将它的名字设置为 **Trailer** 报头的值。下面就是 **Trailer** 的一个可能值：

Trailer: Content-Length

服务器会在提供完实体消息体并知道了所提供的数据的大小以后就为 **Content-Length** 提供一个值并添加到实体消息体后面。

Transfer-Encoding

339

类型：响应报头。

重要程度：低。

Transfer-Encoding 的作用和 **Content-Encoding** 是相同的：将一些临时性的转换（transform）应用到实体消息体上（通常是压缩），而这些转换会在另一端以透明的方式还原回去。它们的区别就在于“另一端”，**Transfer-Encoding** 的“另一端”比 **Content-Encoding** 的“另一端”离服务器的距离更近。

考虑一个这样的场景：HTTP 客户端通过一个代理与服务器进行通信。就 **Content-Encoding** 而言，转换的两端分别是服务器和客户端。但是就 **Transfer-Encoding** 而言，有两个转换会发生：一个是客户端与代理之间的转换，一个是代理与服务器之间的转换。

如果服务器对实体消息体进行了压缩，并设置了 **Content-Encoding: gzip**，代理（很可能）将不处理实体消息体而还是以压缩后的形式将它发送出去。而如果服务器设置的是 **Transfer-Encoding: gzip**，那么代理的任务就是对实体消息体进行解压，然后将其发送出去。

在 TE 中，这个报头的值被称为一个“传输码”，而 IANA 也在 <http://www.iana.org/assignments/http-parameters/http-parameters.xml> 保存了一份合格的传输码注册表。大部分传输码指的是压缩算法，并且可以作为 Content-Encoding 的值使用，但是有一个值只能用于 Transfer-Encoding，它就是：chunked。

有时候，服务器需要在不知道一些重要信息比如大小的情况下发送实体消息体。与其去掉一些依赖那些信息的 HTTP 报头，比如 Content-Length 和 Content-MD5，服务器可以决定将实体消息体一块一块地发送出去，最后再将 Content-Length 等类似的报头追加到实体消息体的后面，而非前面。服务器发送 Transfer-Encoding: "chunked" 就是声明它将要这么做。在所有的数据块都发送出去以后，服务器就知道了那些它之前不清楚的信息了，它可以将 Content-Length 和 Content-MD5 作为“trailer”而非“header”进行发送。

HTTP1.1 要求客户端支持分块的传输编码，但是许多可编程的客户端并没有支持这一功能。

Upgrade

类型：请求报头。

重要程度：低，将来可能高。

340 如果你更希望使用一些其他的协议而非 HTTP，你可以通过发送 Upgrade 报头来告诉服务器。如果服务器恰好支持这个协议，它会发送回一个响应码 101 (Switching Protocols)，然后立即开始采用新的协议。

RFC 2817 制定了一个包含了 Upgrade 报头的可能值的 IANA 注册表。现在这里面只有 3 个值：HTTP、TLS/1.0（也就是 HTTPS）和 WebSocket。

除去 WebSocket（一种用于 web 浏览器的但不符合 REST 范式的协议）不说，Upgrade 报头现在还并不经常被使用。想要使用 HTTPS 的客户端会一开始就使用 HTTPS。但是在 HTTP2.0 标准完结之后，而客户端可以默认任何给定的服务器都支持 HTTP2.0 之前的这段时间里，Upgrade 报头可能会变得相当流行。

User-Agent

类型：请求报头。

重要程度：高。

这个报头能够让服务器知道是哪种软件在发起 HTTP 请求。在人类 web 中，它是一个标识浏览器品牌的字符串。在 API 的世界中，它通常标识的是编写客户端所采用的 HTTP lib 库或者客户端 lib 库。它也可以标识一个具体的客户端程序。

在 Web 流行后不久，服务器就开始通过读取 User-Agent 来判断另一端的浏览器的类型。它们会根据 User-Agent 的值来发送不同的表述。这是一个可怕的主意。User-Agent 检测的方案不仅会使得 web 浏览器间的不兼容性问题长期存在，而且会导致增加许多新的 User-Agent 值。

几乎目前所有的浏览器都伪装成 Mozilla，因为它是第一款流行的浏览器（Netscape Navigator）的内部代码名称（internal code name）。一款不伪装为 Mozilla 的浏览器可能不能得到它所需要的表述。还有一些浏览器同时伪装成 Mozilla 和 Internet Explorer，这样它们就可以触发那些最初只是运行在 Internet Explorer 浏览器上的代码。一些浏览器甚至允许用户为每次请求选择 User-Agent，以此来欺骗服务器发送正确的表述。这真是糟糕透了。

不要让历史重演。API 应该仅仅使用 User-Agent 来统计需不需要和拒绝某些编码质量很差的客户端。它不应该使用 User-Agent 来为特定的客户端修改表述。这同样适用于其他用于识别某个特殊软件 agent 的方式，比如 OAuth 客户端证书。

Vary

类型：响应报头。

重要程度：低到中等。

Vary 报头告诉客户端，它可以修改哪些请求报头来获得某个资源的不同的表述。如下例：

```
Vary: Accept Accept-Language
```

这个值告诉客户端，它可以通过设置或者修改 Accept 报头来请求不同数据格式的表述，还可以通过设置或者修改 Accept-Language 来请求不同语言的表述。

这个值同样也可以用来告诉缓存让其“将某个资源的日文表述与英文表述来分开进行缓存，即使这两个表述拥有相同的 URL”。日文表述并不是一个全新的字节流，它不会使所缓存的英文版本的表述失效。两个请求可以为 Accept-Language 报头设置不同的值，所以它们所对应的表述也应该分开缓存。

如果 Vary 的值是 *，那么它的意思是，表述不应该被缓存。

Via

类型：请求和响应报头。

重要程度：低。

当某个 HTTP 请求直接从客户端发送到服务器，或者响应直接从服务器端发送到客户端时，这里面并没有 **Via** 报头。当它们之间存在一些中间媒介（比如代理）时，每个媒介都会在请求或者响应消息中添加一个 **Via** 报头。消息的接收方可以查看 **Via** 报头来了解这条 HTTP 消息通过中间媒介的路线。

Warning

类型：响应报头（从技术上讲也可用于请求）。

重要程度：低。

Warning 报头是对 HTTP 响应码的一个补充。它通常由中间媒介比如缓存代理来添加，以告知用户那些可能存在但是不能从响应中明显看出的问题。

就像响应码那样，每个 HTTP **warning** 拥有一个由 3 个数字组成的值：“警告码 (warn-code)”。大部分警告都与缓存行为有关。下面的这个 **Warning** 是说，尽管 **localhost:9090** 上的缓存代理知道缓存的响应是过期的，但是还是会将其发送出去：

342 > Warning: 110 localhost:9090 Response is stale

警告码 110 的意思是“响应过期了 (Response is stale)”，就像 HTTP 响应码 404 的意思是“没有找到 (Not Found)”一样。HTTP 标准定义了 7 个警告码，我在这里就不再介绍了。

WWW-Authenticate

类型：响应报头。

重要程度：非常高。

这个报头是和响应码 401 (**Unauthorized**) 一起出现的。它表示的是服务器要求客户端在下次请求这个 URI 的时候要发送一些认证信息。它也会告诉客户端服务器所期待的认证类型。而这很可能是 HTTP Basic Auth 或者某个版本的 OAuth。

为API设计者准备的Fielding论文

导读

一个 RESTful 的系统应该要遵守一系列的原则，我在本书中贯穿始终地使用了“Fielding 约束”一词来作为这些原则概念上的缩写，并且谈论着“无状态”、“超媒体约束”等词汇。该附录是我的一次尝试，我试图采用一种略为正式的方式来在本章的内容中对这些词的意思，以及它们之间是如何相互作用的进行说明。

Fielding 约束是在 Roy Fielding 的博士论文^{注1}中定义的 Web 的“架构属性”。对于一般的开发者来说，理解那些深奥的学术风格的论文的推理过程以及高度抽象的操作步骤要比理解一篇 RFC 要困难很多。所以让我首先来展示下由 Fielding 的工作所带来的实践价值。

Roy Fielding 在 20 世纪 90 年代花费了大量的精力来进行 HTTP 协议 1.0 版本（见 RFC 1945）的正式化工作，并开发了 HTTP 1.1 版（该版本最终成为了著名的 RFC 2616）。Web 已然是一个巨大的成功，但是它的成功也显露了一些设计问题，这些问题曾经可能会阻止 web 进一步扩展，从而令它无法达到我们今天所享有的水平。

Fielding 论文展示了一个类 web 系统可能具有的若干属性，并遴选出了那些可以促使 Web 成功的属性。然后该论文选择了一些可以让通常的网络系统看起来比较像 Web 的约束，即 Fielding 约束。这些约束就是 REST：一个抓住了 Web 本质的正式的架构定义。

注1 Fielding, Roy Thomas 在加州大学尔湾分校 2000 年的博士论文《*Architectural Styles and the Design of Network-based Software Architectures*》（该论文的中文版由李锐组织翻译，中文名为《架构风格与基于网络应用软件的架构设计》，本章涉及到论文原文的翻译部分大都参考或引用了该译文，后续不再一一注明——译者注）。

乍一看，这是一个不切实际的想法。现实的 Web 并不出自于某个架构定义。它是由众多物理学家和黑客们共同铺就而成的。它本应该毫无理由地符合某个计算机科学家的理论框架。你猜怎么着：它实际上并不符合！在由 Fielding 约束描述的理想 web 和 20 世纪 90 年代中期现实生活中的 Web 之间存在着大量的断点（disconnects）。“资源”的原始定义过于专注于静态文档。而流行的 HTTP 扩展“cookies”（最初由 RFC 2109 定义）却导致了巨大的问题。服务器有时会得到一些它们不知道应该如何处理的有效请求。

这就是 Fielding 理论能够回报我们的地方。Fielding 约束和现实生活中的 Web 之间存在着断点，但是这并不意味着它的模型是没有用的。这些断点指出了问题所在！修复它们将可以使得现实中的 Web 更加接近于由 REST 描述的理想 web：一个没有可伸缩性问题的 Web。

HTTP 1.1 (RFC 2616) 和 URI 标准 (RFC 2396) 修复了大部分理论与实践之间的断点。通过使用 Fielding 约束作为蓝图，Web 得到了修复。而那些没有修复的断点，例如 HTTP cookies，时至今日仍然导致着各种问题。

我认为“REST”背后的思想是很重要的，因为 web API 与 20 世纪 90 年代的 Web 正处于差不多相同的境况。我们拥有一堆胡乱堆砌的系统，这些系统大都是出于一时权宜的考虑而设计的，而并非出于对可伸缩性和长期的可维护性的考量。Fielding 约束指明了一条通向更加美好的世界的路。将我们现有的 web API 与一组理想化的原则进行对比，可以显现出那些需要我们去修复的问题。

因此让我们来看看 Web 的架构属性；即 Fielding 约束尝试捕获的那些东西。

本附录中的所有引用都来自 Fielding 论文。我同样也推荐阅读 Fielding 在 2008 年的一篇博文，“REST APIs must be hypertext-driven” (<http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>)。

Web的架构属性

Fielding 论文的第 2 章展示了一堆关于一个网络化系统可能拥有的“架构属性”：性能、简单性、可靠性等。它们全都是像母爱和苹果派（motherhood-and-apple-pie，表示纯粹和完美）一样的东西。没有人会公然地反对“简单性”或“可靠性”。

在第 4 章中，Fielding 做出了艰难的选择。这一章识别出了万维网的 4 个关键的架构属性。是这些属性造就了 web 的成功，这也是 Fielding 愿意为此牺牲其他属性的原因。

低门槛

参与创建和构造信息都是自愿的，因此采用“低门槛”策略是十分必要的。

Web 之所以能跑起来是因为它使用起来足够简单。学习如何使用 FTP 或 Telnet 需要记忆大量晦涩的命令。但是当你在启动 web 浏览器后，你看到的是人类可读的文本，并且贯穿这些文本，你都能看到链接到相邻 web 页面的链接。每一个链接都包含了一定的上下文，从而帮助你决定接下来要点击哪个链接。当你点击了某个链接之后，刚才的过程将会重新开始。

◀ 345

相比于之前的 web 超文本系统，构建一个网站同样也非常地容易。你无须使用特殊的编辑程序来编写 HTML 页面；你只需要一个文本编辑器。

可扩展性

简单性使得部署某个分布式系统的最初实现成为了可能，可扩展性使得我们能够避免永远陷入已部署系统的局限之中。即使有可能建造一个能完美匹配用户需求的软件系统，那些需求也会像社会变革一样时间而发生变化。如果一个系统想要像 Web 那样“长命”，它就必须做好应对变化的准备。

脱离了可扩展性，一个系统将只能部署一次。只要用户高兴（他们一开始将会很高兴），那么一切都没问题。一旦用户的需求改变，他们将会转换到另一个不同的系统。

Web 已经让用户们保持快乐了 20 年。在这 20 年间，市值数十亿美元的互联网帝国沉浮更替，而它们全部都构建于相同的四种技术之上：HTTP、URI、HTML 和 JavaScript。

分布式超媒体

分布式超媒体使得表达和控制信息可以存储在远程地点。

在任何客户 - 服务器系统中，服务器掌管着数据集。这就是“存储在远程地点”。客户端可以尝试更改数据集，但是这些更改总是受制于服务器的许可。

分布式超媒体原则采取了一些说明性的指示，是关于你可以对数据做何处理的指示（“表达和控制信息”），你可以将它作为数据本身一样对待。服务器负责着所有的一切。

Web 服务器使用 HTML 文档来传达资源状态，并以此来宣告资源之间的链接关系（安全的转换），同时也宣告了允许用于修改资源状态的机制（不安全的转换）。客户端从服务器获取超媒体文档并读取文档中的全部这类信息，这些文档会随着系统的改变而改变。

那么“表达和控制信息”去了哪里呢？它们可以被编程到客户端中去，或者以人类可读

的文档形式被完全保存在系统之外。这就是今天大部分 API 的做法。这两种技术使得要在不破坏客户端的情况下改变服务器的工作方式变得非常困难。然而，如果你无法改变服务器，你将不再具有可扩展性。

互联网规模

“互联网规模”听上去是一个很时髦的词，意味着“真的很大”，但是 Fielding 考虑的是两个特定的条件。首先，“无法控制的可伸缩性”，它否定了系统间不同部分之间会保持长期关系或协调这一思想。

不能期望客户保持所有服务器的信息，也不能期望服务器跨多个请求保持状态信息。超媒体数据元素不能保持“回退指针”（back-pointers，即用于引用这些数据元素中的每一项的标识符），因为对一个资源的引用数量与对此信息感兴趣的人数是成正比的。

其次是“独立部署”，因为系统间的不同部分没有了长期的关系，所以它们会以不同的程度发生变化。

多个组织边界也意味着系统必须准备好应对逐渐的和片段的改变，旧的组件实现将会与新的组件实现共存，而不会妨碍新的组件实现使用它们的扩展功能……架构作为一个整体，必须被设计为其架构元素易于以一种部分的、迭代的方式来部署，因为强制以一种整齐划一的方式来部署是不可能的。

API并不（相当于）是Web

正如 Fielding 所见，这便是 Web 的 4 个关键架构属性。显而易见，其中的一些是适用于 web API 的。在其他条件都相同的情况下，我们更偏向于选择一个能扩展的系统而不是一个不能随时间变化的系统。即使是一个很小的 API，如果它位于公共的互联网上，那么它就是“互联网规模”的，因为它所面对的问题来自无法控制的可伸缩性和独立部署。

但是我们不能假设这些原则可以被直接搬到 web API 的世界里，这是一个错误。在 Web 和 web API 之间存在着一个很大的决定性差别：语义鸿沟。这一差别颠覆了 Web 的架构属性之间的关系，并迫使我们从这 4 个理想的属性中做出选择。

当有人来做出所有的决策时，“分布式超媒体”是用于降低“门槛”的最简单的方式。在那一刻，某个人可以看到所有可能的状态转换并从中选出一个。但是一旦在这个决策的循环中脱离了人类，程序员必须创建一个可以填补这个“缺失的人类角色”的软件程序。在这种形势下，“门槛”就被“分布式超媒体”提升了。超媒体文档将每个问题切分成小块，然后创建一个需要对整个问题空间有所理解的决策机器人。

为 web API 降低“门槛”的最简单的方式就是去除“分布式超媒体”属性，并用人类可读的行文格式提前对系统进行描述。

而这样做带来的问题是，“分布式超媒体”属性是唯一能够带来“可扩展性”属性的东西。改变人类可读的文本将无法改变客户端对于系统视图的匹配。没有了“分布式超媒体”，也就没有东西可以保证能改变客户端的系统视图。“互联网规模”表明有太多的客户端需要去跟踪，而未升级的客户端将可能在很长一段时间内停滞不前。

当那些客户端包含了用于取代“分布式超媒体”的硬编码信息时，你便丧失了“可扩展性”。一开始，你的 API 看上去貌似的确是你的用户想要的，但是当他们的需求改变时，他们将渐行渐远，因为你无法进行改变来留住他们。

在 Web 上，这 4 个架构原则互相促进稳固。但是在 web API 的世界里，它们却产生了一些冲突。下面是用于解决这些冲突的 3 种方式：

- 如果你有办法强制所有客户端都同步地进行升级，那么你可以放弃“互联网规模”。这样你就有了“低门槛”和“可扩展性”，而不需要使用“分布式超媒体”。这是将一个 API 部署在一个公司内的通用选择。
- 如果你需要“互联网规模”，你可以为了“低门槛”而放弃“可扩展性”和“分布式超媒体”。当今的大部分 web API 都是这么做的。
- 或者你可以拥抱“分布式超媒体”，并以较高的门槛作为代价获取“可扩展性”和“互联网规模”。这就是我在本书中所采用的方式。我在 ALPS 和通用 profile 上进行的工作，就是一种为了降低超媒体 API“门槛”的尝试。

接口约束

现在让我们来看看 Fielding 约束，这是能够给予万维网理想架构属性的规则。这 4 个最著名的 Fielding 约束可以从论文第 5 章的一段评论中找到：

REST 由 4 个接口约束来定义：资源的识别 (identification of resources)、通过表述来操作资源 (manipulation of resources through representations)、自描述的消息 (self-descriptive messages) 以及作为应用状态引擎的超媒体 (hypermedia as the engine of application state)。这些约束将在 5.2 节中进行讨论。

这些约束组成了 REST 的“统一接口”。它们确实在论文的 5.2 节中进行了讨论，但是并没有以一种你所期望的方便的列表格式。在这一节里，我将会分别对它们进行讨论。

资源的识别

传统的超文本系统……使用随信息的变化而改变的唯一节点或文档标识符，并依赖链接服务器 (link server) 以独立于内容的方式来维护引用。因为集中式的链接服务器完全无法满足 Web 的超大规模和跨越多个组织领域的需求，所以 REST 采用了其他方式——依赖资源的创作者来选择最符合被标识的概念本质的资源标识符。

“资源的识别”是 Fielding 所使用的名字，而我把它叫作“可寻址性 (addressability)”。一个 URI 标识一个资源。资源的状态可能会发生改变，但是它的 URI 将保持不变。如果资源的 URI 发生了变化，服务器会使用超媒体（即 Location 报头）来将客户端引导至新的 URI。

在 Web 占主导地位的今天，很难想象那些“传统超文本系统”会一直更改它们的标识符。但是不难想象存在着另一个问题：网站和 API 将过多的资源状态分配给了一个单一的 URL，就像我在第 1 章中所抱怨的那个餐馆网站。

通过表述来操作资源

REST 组件使用表述来捕获某个资源的当前状态或预期状态，随后在组件之间移交该表述，通过这种方式在资源上执行各种动作。表述由一个字节序列和描述这些字节的表述元数据构成。

Web 对“资源”这一概念所采用的视角是很宽泛的。资源可以是任何事物。这意味着资源是在互联网上发送的任何物理对象或抽象概念。尽管如此，我们可以使用表述来谈论这些资源。

一个表述是一个“字节序列”，所以它可以在网络上传输。它“捕获了资源的当前状态或预期状态”，所以客户端可以将它作为真实事物的替身来使用。而且表述并不与产生它的服务器端代码绑定，这意味着当服务器实现发生改变时表述是无须改变的。

在 Web 上，客户端和服务器端使用了一小组标准化的 HTTP 方法 (GET 和 POST)，并通过将表述来回地发送从而对资源进行操作。一个 web API 可以添加更多的方法 (PUT、DELETE 等)，但是这仍然是一个很小的集合，并且需要扩大社区对此的共识。在客户端和服务器端之间丰富的交互几乎完全可以在它们互相发送的表述中找到。

自描述的消息

REST 通过强制要求消息具有自描述性来支持中间组件的处理，其具体体现为：请求之间的交互是无状态的、使用标准的方法和媒体类型来表达语义和交换信息，以及响应可

以明确地表明其可缓存性。

一个 HTTP 消息包含了所有足以让接收者理解它的必要信息。即使没有围绕该消息文档可供参考，客户端也能理解这些消息。如果理解一个消息需要理解一些其他的文档，比如媒体类型定义或一个 profile，那么该消息应该要使用 Content-Type 或 Link 报头来链接到该文档。

让我们来逐个看看 Fielding 论文的例子：

1. “请求之间的交互是无状态的”。无状态约束（下面会谈到的）只是自描述消息约束的一个特例。在一个无状态的系统中，服务器可以独立处理每个请求，而无须对该客户端先前所有请求的处理进行记忆。
2. “使用标准的方法和媒体类型来表达语义和交换信息。”非常简单，如果 HTTP 响应不包含 Content-Type 报头，客户端将不知道如何来解析实体消息体。如果 HTTP 请求没有提到使用的 HTTP 方法，或者编造了自己的方法，服务器将不知道如何处理该请求。
3. “响应可以明确地表明其可缓存性。”客户端只是从 web 服务器得到了一个 HTTP 响应。客户端对该响应进行缓存是否有意义？如果有意义，那么该缓存多久呢？一分钟，或者是一年？

客户端不应该来做这个决定。服务器所处的位置更适合知道响应可以被安全地缓存多久。因此，需要服务器来担负起给到客户端缓存信息的工作。

现在我们知道了，这就是自描述消息的出处。在 HTTP 中，服务器通过向特定的可以被缓存的 HTTP 响应添加报头来传达缓存信息。服务器没有带外通信来说明如何缓存它所发送的缓存消息。缓存指令是消息的一部分，它们与消息一起被缓存。当从缓存中重新获取消息时，需要检查该消息是否还没过期，客户端可以基于消息中的信息来做出决策。

较早的 HTTP 版本没有达成“自描述消息”这一理想。Fielding 论文的 6.3.2 一节讨论了由此产生的一些问题。最为典型的是，没有 Host 报头，服务器就没有办法知道应该由哪个域名来处理这个传入的 HTTP 请求。这使得在一台机器上托管多个域名变得非常困难。

即使在 HTTP 1.1 中，如果一个响应消息没有包含任何能将其与原始请求联系起来的说的话，是不满足自描述消息约束的。与此相比，CoAP 协议是使用 token 和消息 ID 来将响应与请求联系起来的。HTTP 2.0 将可能采用相同的方式。

超媒体约束

Fielding 论文从未显式定义过这个声名狼藉的习语“将超媒体作为应用状态引擎”，但是如果你理解以下个别概念，你应该能搞清楚它的意思：

1. 所有的应用状态维持在客户端一侧。改变应用状态是客户端的职责。
2. 客户端仅能够通过发起 HTTP 请求和处理响应来改变应用状态。
3. 客户端如何知道接下来可以发起哪些请求？只能通过查看已经收到的表述中的超媒体控件。
4. 因此，超媒体控件是应用状态变化背后的驱动力。

超媒体约束并非是你实现“RESTful”时必须要做的无聊之事，它是你遵守其他约束后产生的结果。它给了你扩展性。超媒体约束允许一个智能的客户端可以自动化地适应服务器端的变化。它也允许服务器可以在不破坏它所有客户端的情况下更改它底层的实现。

选择超媒体作为用户界面 (user interface)，是因为它的简单性和通用性：无论信息来源于何处，都能够使用相同的界面；超媒体关系（链接）的灵活性允许对其进行无限的构造 (allows for unlimited structuring)；对于链接的直接操作允许在信息内部建立复杂的关系 (allows the complex relationships within the information)，来引导读者浏览整个应用。

架构约束

Fielding 论文的第 3 章论述了大量不同且合理的网络架构，并且将它们分解成了它们的“架构属性”：在一个通用的“空风格 (null style)”上的原子的、可互相替换的约束。第 3 章展示了这些原始的约束是如何被结合起来用于描述像分布式对象系统这样的通用架构的。

Fielding 论文著名的第 5 章“表述性状态移交 (Representational State Transfer)”将这种解构的方式应用于了万维网。事实表明，Web 是由来自 Fielding 论文第 3 章中的 5 个属性（“客户 - 服务器”、“无状态”、“缓存”、“分层系统”和“按需代码”）组成的。还有第 6 个属性（“统一接口”），它是由我先前提到的 4 个接口约束构成的。统一接口约束涉及了大多数使得 Web 能独一无二的事物。

351 通常，web API 大都关心“客户 - 服务器”、“无状态”、“缓存”和“统一接口”。而“分层系统”对于 web API 的部署来说较 web API 的设计更加重要。Web API 并不真正使用“按需代码”。即便如此，下面是对所有 Web 的架构约束更为详尽的讨论。

客户-服务器

客户组件通过一个连接器将请求发送给服务器，希望执行一个服务。服务器可以拒绝这个请求，也可以执行这个请求并将响应发送给客户。

对这一项我们想必很熟悉，因为客户-服务器是目前互联网上占据主导地位的网络架构。它甚至会出现你没有想到的地方。很多点对点架构也是客户-服务器架构；这是因为某特定节点有时会扮演“客户”而有时则扮演“服务器”。

客户-服务器架构的主要竞争对手是基于事件的集成架构，在这种架构中，组件会持续地在网络中广播事件，同时也监听着它们感兴趣的事件。在系统中的各个部分（即其中的一方会被认为是“客户”，而另一方则是“服务器”）不存在一对一的通信，这里只有广播和侦听。

无状态

其目的是通过使服务器无须维护当前请求之外的客户端状态，从而改善服务器的可伸缩性。

在 HTTP 服务器看来，在客户端当前没有发起请求时，客户端是不存在的。所有的应用状态，即关于一个特定客户端经过应用的路径信息，是属于客户端的。服务器并不关心。

如果有些应用状态非常重要，以至于需要服务器关心，那么它应该成为资源状态。它应该被创建成一个资源，并具有自己的 URL。这样一来服务器便可以控制该状态，而客户端可以像操作别的资源一样操作该资源。

特别的是，这意味着你不应该在服务器上存储 session ID。引用 Fielding 论文中的内容：

一种形式的滥用是在由超媒体响应形式的表述所引用的所有 URI 中包括标识当前用户的信息。这样内嵌的用户标识可用于维护服务器端会话的状态，通过记录用户的动作来跟踪他们的行为，或者跨多个动作携带用户的首选项……由于违反了 REST 的架构约束，这些系统会降低共享缓存的效率，也降低了服务器的可伸缩性，并且在一个用户与其他用户共享那些引用时会得到不希望的结果。

缓存

这种形式的复制最常出现在可能存在的数据集 (potential data set) 远远超出单个客户端容量的情况下，例如，在 WWW[.] 中。

由于有了自描述消息约束，用于理解响应的所有必要的信息都包含在响应本身之中。由于有了无状态约束，一个 HTTP 请求只需要考虑其自身，且独立于其他的请求。这两项约束使得缓存成为了可能。一个 HTTP 客户端可以自动化地将它的请求与先前接收到的

响应进行匹配，从而尽可能地节省数据在网络上往返的开销。正如 Fielding 所言，“最好的应用性能是通过不使用网络获取的”。

统一接口

实现与它们所提供的服务是解耦的，这促进了独立的可进化性。然而，需要付出的代价是，统一接口降低了效率，因为信息都使用标准化的形式来移交，而不能使用特定于应用的需求的形式。REST 接口被设计为可以高效地移交大粒度的超媒体数据，并针对 Web 的常见情况做了优化。但是这也导致该接口对于其他形式的架构交互而言并不是最优的。

我已经讨论过统一接口。它是由 4 个我先前提到过的接口约束构成的：可寻址性、通过表述来操作资源、自描述消息以及超媒体约束。

Fielding 指出了这些约束是偏向“移交大粒度的超媒体数据”的。一个典型的 Web 浏览器通过向一个 URL（可寻址性）发送 GET 请求，并取回一个充满了链接的大型 HTML 表述（表述的使用）来开始它一天的工作。它的人类用户阅读了文档并访问了其中的一个链接（超媒体约束）。这使得浏览器发送了另一个 GET 请求并获取到另一个充满链接的 HTML 表述。

大多数的 HTTP 请求是 GET 请求，它们执行的是安全的状态转换。这也是为什么有这么多专注于减少 GET 请求开销的 HTTP 性能优化策略（缓存、有条件的请求、局部请求）。

如果 API 设计是重度面向非安全的状态转换的，那么在 Web 统一接口和最能满足 API 客户端需求的接口之间将会存在很大的断点。但是事实证明，API 客户端同样在移交大粒度的数据方面（如果不是超媒体移交的话）花费了它们大部分时间。这就是为什么一个所谓的“RESTful”API 即使没有遵守超媒体约束也可以非常成功的原因。该 API 给它的用户带来了其他 3 个接口约束的好处。

一些超媒体感知的 API 忽略了“移交大粒度的超媒体数据”中的“大粒度”这一部分。为了取代提供传达了大量资源状态的大型文档（例如用于表述一个人的 hCard 文档），它们将信息切分成了多个资源（为一个人的名字、姓氏以及生日分别赋予一个独立的 URL）。客户端必须发起多个 GET 请求来获取它所需要的信息。这样的结果就是产生一个延时很高的“聊天式”API。

这并不是技术上的错误，但是它对性能是有害的。HTTP 2.0 应该会实际支持编写这种类型的 API：一个基于移交较小粒度的超媒体数据的 API。

分层系统

分层 - 客户 - 服务器风格在客户 - 服务器风格的基础上添加了代理组件和网关组件……

这些额外的位于其间起到斡旋作用的组件为系统添加了多个层，可以用来实现诸如负载均衡和安全性检查这样的功能。

贯穿本书，我一直在讨论 HTTP 客户端和 HTTP 服务器，前者是一部分用于发起 HTTP 请求的软件，而后者是一部分用于发起 HTTP 响应的软件。但是 HTTP 规范还定义了其他两类可以置于客户端和服务器之间的软件：代理和网关。所有的这些都是 HTTP 系统中的组件。

代理从组件（客户端、代理和网关）接收 HTTP 请求，就像一台 web 服务器。与 web 服务器不同的是，代理自己并不处理请求，它将请求传递给另一个组件（一台服务器、一个代理或一个网关）并等待响应。当它接收到响应，代理会将该响应传递给向它发送请求的组件。代理可以在传输过程中修改请求和响应：比如压缩数据、消除识别信息或者执行审查机制。

网关是一个用于在 HTTP 和某些其他协议之间进行翻译的代理。网关可以将一个 HTTP 请求转化成一系列命令，然后从一台 FTP 服务器上下载文件，并将下载好的文件作为 HTTP 响应的实体消息体。在客户端看来，它只是发起了正常的 HTTP 请求并得到了 HTTP 资源的表述。

“分层系统”约束很少谈论到代理和网关，更多讨论的是一个事实，即向客户端和服务端之间添加一个组件几乎是一个透明的操作。客户端并不知道它是否在与服务器直接对话，或是在与一个代理对话，而该代理则是在与另一个与网关交谈的代理对话。

客户端、服务器、代理和网关都具有相同的接口，并不存在什么“代理协议”。代理也接收 HTTP 请求，发送 HTTP 响应。唯一存在的是一些与代理相关的特殊状态码（我在附录 A 中提到过）；少量用于控制代理的 HTTP 报头（见附录 B）；以及两个 HTTP 方法，CONNECT 和 TRACE，用于使用和调试代理。但是对于客户端来说，代理看上去就像是一台 HTTP 服务器。而对服务器来说，代理看上去就像是一个 HTTP 客户端。

HTTP 确实定义了大量用于指导“分层系统”架构元素与“缓存”元素该如何互相作用的复杂规则，但是伴随着代理组件的出现，这一切都落幕了。客户端和服务端无须再处理这些规则。

代理对于现实世界中的 API 部署来说非常有用。代理可以通过将不同的请求分发到不同的服务器从而起到负载均衡的作用。代理可以对被频繁访问的表述进行缓存，这样一来客户端请求将无须每次都到达服务器。但是我不会在本书中详谈代理和网关，因为根据分层系统的整个前提来说，它们都是不可见的。另一个 Fielding 约束，特别是无状态，使得在客户端和服务端之间添加和去除大型的中间组件链，而同时又不被客户端和服务

◀ 354

器察觉成为了可能。

如果想要对代理和网关的处理进行更加详尽的了解，我推荐阅读《HTTP: The Definitive Guide》^{注2}一书的6章和第8章。

按需代码

[A] 客户组件知道如何访问一组资源，但不知道如何处理它们。它向一个远程服务器发送请求，以获取如何处理资源的代码。接收到这些代码之后，在本地执行这些代码……[T] 最大的限制是：由于服务器发送代码而不是简单的数据，因此缺乏可见性。如果客户端无法信任服务器，缺乏可见性会导致明显的部署问题。

按需代码对软件起到的作用就好比超媒体对数据起到的作用。万维网就是以按需代码的方式工作的，但是我不会在本书中详谈这一点，因为对在一个 web API 的环境中使用按需代码我并没有什么好的建议。就我讨论过的那一大堆超媒体格式而言，唯一支持按需代码的只有 HTML。

HTML 的秘密就是 `<script>` 标签，它可以自动地获取某资源的表述，然后将该表述作为 (JavaScript) 代码进行执行。多亏了 `<script>` 标签，从而让访问你网站的用户可以下载并运行一个复杂的软件应用。当应用发生变化后，该人类用户可以通过重新加载 web 页面以及重新下载所有内容来“重新安装”该应用代码。

对于 API 客户端代码来说，目前最常用的部署策略就是使用各种各样的编程语言来编写代码库（为 API 准备的 API，如果你愿意的话），然后为独立的开发者们提供下载。这是个坏主意，原因很类似：无法感知超媒体的 API 都是坏主意，它破坏了可扩展性。客户端代码库终究会发生改变，但是没有人能区分那些现有安装好的代码基础。旧版本的客户端启动并运行它一贯运行的相同代码，但是现在这些代码已经是错误的了。

按需代码可以解决这个问题。有了按需代码，客户端代码库可以在它们发布时自行下载它们的新版本。能够和客户端的编程语言 API 保持一致，基于客户端代码库的代码可以在底层实现改变后继续工作。

现在的问题是，没人真喜欢从别人的服务器上自动下载以及运行代码。如果提供超媒体文档的服务器受到了安全威胁，它的客户端将可能得到伪造的数据。这非常不好，但是这可能会更糟。如果一台提供按需代码的服务器受到了安全威胁，那么它的客户端也可能遭受到安全威胁。

按需代码之所以可以在 Web 上工作，是因为 web 浏览器是在一个沙箱中运行下载的

注2 该书的中文版《HTTP 权威指南》已经由人民邮电出版社翻译出版。

JavaScript 代码的。即使如此，按需代码还是导致了一些例如跨站脚本攻击这样的浏览器安全问题。通常来说，自动化客户端一般都不会运行在沙箱中。它们需要访问本地文件系统、数据库以及其他的系统资源。糟糕的按需代码可以对系统造成很大的破坏。

在一个所有参与者都互相信任的环境中（例如在一个公司中），使用按需代码来进行部署是可行的。但是针对 RESTful 架构来说，这样的环境是最为脆弱的，而针对如何在一台可信服务器上为客户端部署软件，存在着其他可行的方式。

因为所有的这些原因，我不期望按需代码会在任何时候替代可下载的客户端（downloadable client）。一个超媒体感知 API 的可下载客户端在 API 改变时最不可能被破坏，这使得它在脱离按需代码后继续存在成为了可能。

总结

Web 的成功来自以下 4 个架构属性：

低门槛

学习使用 Web 非常简单，构建网站也非常简单。

可扩展性

独立的网站可以突然发生变化，但不会破坏它们的客户端。经历了数十年的时间，Web 发生了翻天覆地的变化，但是底层的技术却并没有太多变化。

分布式超媒体

有关客户端可以对服务器的数据做何处理的信息和这些数据一起放在相同的地方，并且在同一份文档中被发送到了客户端。

互联网规模

系统和它的每个组成部分之间不存在长期的关系，不同的组成部分可以按照不同的程度发生变化。

通过以下 6 个架构约束，以上的架构属性已经在 Web 上得到了实现：

客户 - 服务器

Web 上的所有通信都是一对一的。

无状态

在客户端当前没有发起请求时，服务器并不知道它的存在。

356 缓存

客户端可以通过复用缓存中早前的响应来节省数据在网络中来回传输造成的开销。

分层系统

像代理这样的中间组件可以被透明地插入到客户端与服务器之间。

按需代码

除了数据之外，服务器可以发送可执行的代码。当客户端请求该代码时，该代码可以自动化地进行部署，如果代码发生了变化，它也可以自动化地进行重新部署。

统一接口

这是一个涵盖了4个接口约束的术语。

资源的识别

每个资源都由一个稳定的URI进行标识。

通过表述操作资源

服务器通过向客户端发送表述来描述资源状态。客户端通过向服务器发送表述来操作资源状态。

自描述消息

用以理解请求或响应消息的全部必要信息都包含在（或至少有链接来自）消息本身内部。

超媒体约束

服务器通过发送包含了可供客户端自由选择的选项的超媒体“菜单”来操作客户端的状态。

这9项（或者是10项，取决于你怎么计数了）约束便是Fielding约束。

结论

如果语义鸿沟不存在，那么设计一个 web API 与设计一个网站几乎很相似。我们可以盲目地复制 Web，而无须真正理解它是怎么工作的。我们将不需要 Fielding 论文，并且无须根据 Fielding 约束来评价我们的 API，因为我们已经有了一个关于如何创建我们想要的系统的成功实践的例子。

Web 已经几乎足够好，从而能满足我们的需求，但也不完全是。我们想要将工作建立在它的成功之上，但是我们不能使用同样的原则。解决方案就是 Fielding 论文中经常被忽视的前半部分，它展示了 Fielding 约束是怎么得到的。它们来自如下这样一个过程：

1. 写下所有可能会为你的系统带来好处从而值得拥有的架构属性。
2. 识别出哪些属性才是你真正需要的，而哪些是你愿意牺牲掉的。
3. 得出一组将会为你的系统带来你真正需要的架构属性的架构约束。
4. 设计一组协议和其他的标准，它们可以协同工作，从而体现这些约束（HTTP、URI、HTML、JavaScript）。
5. 经历几十年的过程之后，当问题开始显现，便开始重复步骤 2 ~ 4（HTTP 0.9、HTTP 1.0、HTTP 1.1 和即将到来的 HTTP 2.0）。

357

Web API 引入了新的约束：语义鸿沟。我们一直在围绕着语义鸿沟进行着争论，已经长达十年之久，即使我们已经达成了共识，一部分 Fielding 约束仍然是适用的，但是我们仍然看不到解决方案。看看你持的是哪一方的观点，你必须判断哪些 Web 架构属性对你来说才是最重要的。

是低门槛吗？该属性能让人们无须进行针对特定网站的培训也能使用网站。是可扩展性吗？该属性能让一个网站经得起一次全新设计而不会破坏它的客户端。又或者是互联网规模？该属性允许每个人都使用他们自己选择的 web 浏览器并根据他们自己的步调进行升级。

我选择了可扩展性和互联网规模，而牺牲了低门槛。这是因为语义鸿沟本身提升了门槛。任何 API 都比具有相同意义的网站更难使用。一个基于超媒体的设计提升了门槛，但是它给了你可扩展性和互联网规模，对于长期来说，这是必不可少的。

而且这是一个长期的项目。我们可以通过采用更智能的客户端代码库，采纳使用常见的超媒体类型和常见的应用语义来重新获得较低的门槛。但是如果放弃了可扩展性和互联网规模，那么我们将再也无法重新得到它们。

应用语义 (application semantics)

一个表述的应用语义就是指用现实世界中的概念来解释该表述所对应的资源。两个使用完全相同的标签的 HTML 文档可以具有完全不同的应用语义——其中一个描述的可能是某个人，而另一个可能描述的是一个医疗流程。

如果某种文档格式是设计用来表现现实世界的概念的，我们可以说这种格式本身就拥有应用语义。Maze+XML 格式就拥有用于表现迷宫游戏所必需的应用语义。而 HTML 格式的应用语义就是人类可读文档。HAL 格式则没有值得称道的应用语义：每个用户必须提供他们自己的应用语义。

术语“应用语义”是为本书而虚构的。它并不是一个标准术语。

应用状态 (application state)

客户端在执行 API 时的轨迹信息就是应用状态。大部分客户端都是以相同的状态从 API 的“主页”开始的。随着它们做出不同的决策，触发不同的超媒体控件，最终会停在不同的地方，它们的应用状态也就越离越远了。

缓存 (cache)

HTTP 响应的存储库，用于提高客户端性能。客户端有时候可以复用某个缓存的响应而不再通过网络发送新的请求。

按需代码 (code on demand)

Fielding 约束之一。它是说，除了数据之外，服务器还可以发送可执行代码。这些代码会自动部署到客户端，并且能够随着服务器的其他实现的改动而一起改变。由于安全问题，API 很少实现这一约束。

我对超媒体约束的另一种称谓 (参见“超媒体约束”)。我更喜欢这个术语,这是由于它重点突出了我所重视的那部分内容:资源都是通过安全的状态转换相互“连接的”。

非关联化 (dereferencing)

某种将某个 URL 转换为一个表述的计算机化过程。对于 http: URLs, 非关联化意味着向该 URL 发送一个 HTTP GET 请求。

内嵌链接 (embedded link)

它是一种链接,在被触发后,添加到客户端的应用语义而不是代替当前的应用语义。内嵌链接经常被自动触发,比如 HTML 的 `` 和 `<script>` 标签。可对比转出链接 (outbound link) 进行理解,另见 transclusion。

实体消息体 (entity-body)

一个 HTTP 请求或响应所关联的文档。通常,该文档是一些资源的表述。

报头 (header)

一个 HTTP 请求或者响应所关联的键值对。

HATEOAS

“hypermedia as the engine of application state”的首字母缩写。

超媒体是服务器发送给客户端的数据。它说明了客户端下一步所能执行的操作。HTML 链接和表单都是超媒体。RESTful API 的本质特征就是遵循超媒体约束:它的表述包含了用于描述可能的状态转换的超媒体控件。

将超媒体作为应用状态的引擎 (hypermedia as the engine of application state)

Fielding 约束之一。我将它简称为“超媒体约束”。服务器端通过发送包含超媒体“菜单”来操纵客户端的状态。这个“菜单”包含了客户端所能自由选择的下一步操作。

超媒体控件 (hypermedia control)

一个超媒体控件描述的就是一个状态转换。在 web API 中,一个超媒体控件通常有两部分组成。最重要的一部分是客户端可以发起的 HTTP 请求 (或者请求集合) 的描述。另一部分重要性稍微弱一些的就是链接关系,它说明了在客户端在发起那个 HTTP 请求后所将发生的状态转换。

有一些超媒体控件应该被自动触发 (比如 HTML 的 `` 标签)。另外一些只有在客户端决定触发以后才会被触发 (比如 HTML 的 `<a>` 标签)。

HTTP 方法 (HTTP method)

也被称为“HTTP 动词”。作为 HTTP 请求的一部分,它会在一个非常基本

的层次上告诉服务器，客户端想要对某个资源做什么。

幂等性 (idempotent)

一个幂等的状态转换不管是被触发一次还是多次，效果都是一样的。HTTP 方法 PUT、DELETE、LINK 和 UNLINK 都被认为是幂等的。客户端可以在不可靠的网络中反复执行这些方法，直到它们正常执行完成。

任何安全状态转换也都是幂等的。

信息资源 (information resource)

一种以二进制比特流为原生形式的资源，它不同于物理对象或者抽象概念。信息资源可以作为自己的表述来提供服务。

链接关系 (link relation)

和某个超媒体控件关联的字符串。链接关系说明了在客户端触发控件之后所可能发生的状态转换。链接关系可以描述应用状态的变化 (比如 next 和 previous)，也可以描述资源状态的变化 (比如 edit)。

RFC5988 定义了两种类型的链接关系：扩展关系 (以 URI 形式表现) 以及注册关系 (必须是在某些地方“注册”过能避免冲突的简短字符串)。

媒体类型 (media type)

媒体类型 (也被称为内容类型 (content type) 或者 MIME 类型 (MIME type)) 是用于标识某种文档格式的

简短字符串。一旦你知道了某个文档的媒体类型，你就能解析它了。或许你也能够就此理解它的应用和协议语义。

转出链接 (outbound link)

它是一种超媒体控件。它被触发以后会用一个全新的状态来代替客户端当前的应用状态。HTML 的 `<a>` 标签包含一个转出链接。对比 embedded link。

overloaded POST

通过使用 HTTP POST 方法来触发状态转换的一种用法，这个状态转换可以做任何事情。可对比 POST-to-append 进行理解。

POST-to-append

通过使用 HTTP POST 方法来在某个资源的下一级创建新的资源的用法。可对比 overloaded POST 进行理解。

profile

Profile 对文档的媒体类型所没有覆盖到的语义进行了说明。Profile 就像一副神奇的眼镜，你它可以向你揭示出文档中那些你之前看不到的意义。

比如 hCard profile 可以将一份普通的 HTML 文档变为某个人的简介。在 HTML 标准中并不存在任何描述人类信息的内容。这个 profile 完成了这项额外的工作。

不理解某个 profile 的客户端仍旧可

以根据它对文档的媒体类型的理解来解析文档并从中获取信息。但是它将会丢失一些另外层次上的语义。

协议语义 (protocol semantics)

超媒体控件会讨论客户端所能发起的某个 HTTP 请求 (或者请求集合)。这些就是它的协议语义。它们会告诉你在这种状况下, 哪些 HTTP 协议的子集是有用的。

超媒体控件还可以拥有应用语义。应用语义从现实世界的角度来说明: 哪些信息需要通过 HTTP 请求来提供给服务器、什么事情会发生, 并成为对这个请求做出响应或者客户端应该如何将该响应并入它的工作流中。

当一份文档包含超媒体控件时, 我们会说这份文档本身拥有协议语义。这份文档允许所有由它的超媒体控件所定义的 HTTP 请求。

当一种文档格式考虑到超媒体控件时, 我们会说这种格式本身就拥有协议语义。比如, 我们会说 HTML 的协议语义考虑了 GET 和 POST 请求, 但是并没有考虑 PUT 请求。

术语“协议语义”是为本书而虚构的, 它不是一个标准术语。

表述 (representation)

表述是用来描述某个资源的状态的数据。通常, 一个表述就是一份被作为 HTTP 请求或者响应的实体消息体而被使用的文档。在某些情况下, 将整

个请求或者响应消息作为“表述”来理解会更有帮助。

当服务器向客户端发送某个表述时, 它是在描述某个资源当前的状态。当客户端向服务器发送某个表述时, 它是在试图修改某个资源的状态。

资源 (resource)

万物皆可为资源: 某网页、某个人、这个人的姓名、在特定的某天中这个人的体重、他与另外一个人的关系……任何事物。唯一的限制就是资源必须拥有它自己的 URI。没有 URI, 也就没有什么可聊的了。

客户端从来不会直接与某个资源交互。它只能看到这个资源的状态的描述信息, 这些信息都会写在表述中。

资源状态 (resource state)

表述里布满了资源状态。表述所传递的信息或者是关于资源的当前状态的 (当服务器向客户端发送表述时), 或者是关于资源的所期望的新状态的 (当客户端向服务器发送表述时)。

在 web API 的世界里, 资源状态通常被分成一些离散的数据块 (比如一个人的姓名), 而每个数据块都是由一个语义描述符进行描述的。但是事实上, 这更多是我们编写计算机程序的方式而不是 REST。万维网就不是这样工作的。

资源类型 (resource type)

当你想要讨论资源背后的现实世界的事物或概念（不同于资源的表述中的数据）时，你可以使用一种资源类型。一种资源类型是一个 URI，它将一个资源按照某个抽象的分类目录进行分类，比如 `person`（或者进一步细化是 `http://schema.org/Person` 或者 `http://xmlns.com/foaf/0.1/Person`）。

安全的 (safe)

触发一个安全的状态转换对资源状态的影响效果应该和没有执行任何状态转换的效果是一样的。HTTP 方法 GET、HEAD 和 OPTIONS 都被认为是安全的。

自描述消息 (self-descriptive messages)

Fielding 约束之一。它所说的是，所有用于理解某个请求或者响应消息的含义而必需的信息都包含在消息本身里面（或者至少要从消息中链接过去）。

语义描述符 (semantic descriptor)

一个简短的字符串，它是对资源状态的一些离散数据块进行了命名。语义描述符通常由一个就近的 profile 来提供人类可读的描述，并且不同的 profile 可能会给相同的信息提供不同的名称：比如 “given-name” (hCard)、 “givenName” (schema.org) 和 “firstName” (FOAF)。

术语“语义描述符”是为本书而虚构的。它不是一个标准术语。

语义鸿沟 (semantic gap)

文档结构与它在现实世界中的含义（它的应用语义）之间的鸿沟。媒体类型、机器可读的 profile 以及人类可读的资料都是在以不同的方式来消除语义鸿沟，但是消除这个鸿沟总是需要人类来在某些情况下发明一些新的东西。

术语“语义鸿沟”是为本书而虚构的。它不是一个标准术语。我们把消除语义鸿沟所面临的挑战称为语义挑战。

无状态性 (statelessness)

Fielding 约束之一。无状态约束的要点就是客户端负责所有的应用状态，而服务器负责所有的资源状态。

状态转换 (state transition)

应用状态或者资源状态的改变。链接关系是状态转换的名称。超媒体控件说明了哪个 HTTP 请求将会触发一个特定的状态转换。

嵌入 (transclusion)

内嵌链接将一个表述嵌入到另一个表述中。当 web 浏览器在一份 HTML 文档中遇到一个 `` 标签时，它会为了获取一个二进制图片而发起一个 HTTP 请求并且动态地将呈现的图片插入到 HTML 文档中。并不需要同步保持图片和 HTML 文档；它们

甚至可能会在不同的服务器上。

统一接口 (uniform interface)

Fielding 约束之一。它是对描述了 Web 工作原理的 4 种“接口约束 (interface constraints)”的总称，它们分别是：资源标识符 (identification of resources)、通过表述管理资源 (manipulation of resources through representations)、自描述消息 (self-descriptive messages) 以及将超媒体作为应用状态的引擎 (hypermedia as the engine of application state)。

统一接口约束包含了人们在思考“REST”时所考虑的大部分内容。

URI

用于唯一标识某个资源的字符串。

URL

能够被解引用于获取一个表述的 URI。并不是所有的 URI 都是 URL。我们没有办法解引用 URI `urn:isbn:9781449358063`，所以，它不是一个 URL。

A

- <a> tags, 48, 110
- Accept headers, 239, 319
- Accept-Charset headers, 319
- Accept-Encoding headers, 245, 319
- Accept-Language headers, 239, 320
- Accept-Ranges headers, 246, 320
- Activity Streams, 234
- addressability, 3, 346
- Age headers, 320
- agent-type API clients, 89
- Allow headers, 320
- ALPS (Application-Level Protocol Semantics)
 - application semantics in, 146
 - benefits of, 143
 - examples of, 144
 - lenient format of, 150
 - profile link relation in, 148
 - repository for, 149, 235
- API calls
 - server implementation details and, 67
- API clients
 - automated, 74, 76, 87
 - human-driven, 68–72, 86
- APIs (application programming interfaces)
 - adding hypermedia to existing, 190
 - Collection+JSON in, 21
 - constraints and, 25
 - design of (see design procedure)
 - downfalls of current, xv, 15, 67, 237
 - duplication of effort in, xvi, 26
 - for microblogging, xvii
 - functionality needed in, xvi, 17, 56
 - HTTP GET request in, 18
 - HTTP POST in, 24
 - HTTP response in, 18
 - JSON in, 20
 - life-time assurance of, 188
 - semantic gap in, 27
 - typical documentation in, 122, 134, 144
 - (see also documentation)
 - versioning of, 185–189
 - vs. World Wide Web, 344
 - writing to, 22
- application semantics
 - Activity Streams, 234
 - adding to forms, 119
 - ALPS (Application-Level Protocol Semantics), 235
 - definition of, 357
 - Dublin Core, 234
 - examples of, 27
 - HTML plug-in semantics, 111
 - IANA registry of link relations, 230
 - in ALPS, 146
 - link relations, 232
 - microdata items and, 117
 - microformats and, 114

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

- Microformats wiki, 230
- profiles and, 138
- reusability of, 230
- schema.org, 233
- vs. protocol semantics, 57
- application state, 10
 - definition of, 357
 - examples of, 64
- Architecture of the World Wide Web, Volume One (W3C), 29, 181
- Atom Publishing Protocol
 - AtomPub plug-in standards, 104
 - basics of, 207
 - collection pattern in, 104
 - extensibility of, 104
 - original microblogging standard, xvii
 - shortcomings of, 105
 - Slug header in, 317
 - standards for, 102
 - vs. Collection+JSON, 102
- authentication
 - and statelessness, 250
 - Digest method, 251
 - HTTP Basic authentication, 251
 - registration, 250
 - steps of, 249
 - with OAuth 1.0, 252
 - with OAuth 2.0, 256
 - WWW-Authenticate header, 250
- Authorization headers, 321

B

- Base64, 251
- Basic Auth, 251
- bday class, 113
- Berners-Lee, Tim, 272

C

- cache, definition of, 357
- Cache-Control headers, 242, 321
- caching, in World Wide Web, 350, 353
- canonical URLs, 241
- Çelik, Tantek, 142
- class attribute, 112
- client credentials, 250
- client programming, xxi
- client-server architecture, 5, 45, 349, 353

- CoAP (Constrained Application Protocol)
 - CoRE Link Format, 292
 - delayed response in, 290
 - multicast messages in, 291
 - network layout, 287
 - request-response structure in, 290
 - requests in, 288
 - responses in, 288
 - RESTful architecture of, 293
 - types of messages in, 289
 - uses for, 287
- code on demand
 - definition of, 357
 - in World Wide Web, 352, 354
- Collection+JSON
 - advantages of, 25
 - basics of, 21, 206
 - constraints in, 26
 - document example, 94
 - item representation in, 95
 - search template in, 99
 - semantic gap and, 27, 108
 - standard for, 92
 - vs. AtomPub, 102
 - write template in, 98
- collection-based design
 - adaptation of, 92
 - benefits of, 106
 - DELETE method, 101
 - examples of, 91
 - GET method, 100
 - pagination in, 101
 - PATCH method, 101
 - POST-to-append method, 100
 - protocol semantics of, 100
 - PUT method, 101
 - search forms in, 102
- collections
 - definition of, 93
 - individual resources in, 93
- compression, 245
- conditional request, 242, 248
- connectedness
 - definition of, 357
 - example of, 13
- Connection headers, 322
- content negotiation, 239
- content types (see media types)
- Content-Disposition headers, 322

- Content-Encoding headers, 246, 323
- Content-Language headers, 323
- Content-Length headers, 323
- Content-Location headers, 218, 324
- Content-MD5 headers, 324
- Content-Range headers, 324
- Content-Type headers, 20, 324
- Cookie headers, 325
- cookies, 342
- CoRE Link Format, 292
- corporate standards, xxiv
- crawler-type API clients, 88
- credentials, client vs. personal, 250
- CSS classes, 113

D

- data loss, avoiding, 248
- Date headers, 326
- DELETE method
 - details of, 35
 - function of, 33
 - in collection-based design, 101
- deprecated APIs, 188
- dereferencing
 - definition of, 357
 - URLs vs. URIs and, 50
- description strategy, 263, 266, 286
- design procedure
 - basic steps of, 157
 - database schema vs. state diagrams, 178
 - design advice, 177–190
 - detailed steps of, 158–173
 - examples of, 173, 192–197
 - for existing API, 190–192
 - new media types, 183
 - representation format and, 179
 - resources in, 178
 - standard vs. custom names, 182
 - URL design in, 180
- detail representations, 239
- Digest authentication method, 251
- distributed computing, World Wide Web as, xv, 1
- distributed hypermedia, 343, 353
- documentation
 - ALPS (Application-Level Protocol Semantics), 143
 - application semantics and, 138
 - embedded, 154

- human-readable, 134, 172
- importance of, 197
- improved for new HTTP specification, 238
- JSON-LD (JSON for Linking Data), 151
- link relations and, 139
- linking to profiles, 135
- location of, 134
- profile media type parameter, 136
- profiles for, 135
- protocol semantics and, 137
- semantic descriptors and, 140
- special-purpose hypermedia controls, 136
- unsafe link relations and, 140
- XMDP (XHTML Meta Data Profile) format for, 141

domain-specific designs

- API calls in, 67
- application state in, 64
- bridging the semantic gap in, 83
- collections in, 65
- hypermedia mazes and, 59
- link relations in, 62
- locating, 83
- Maze+XML example, 60
- reusing foundations for, 86
- server implementation and, 72
- standard extensions, 77–80

- draft-ietf-core-coap, 287
- draft-ietf-httpbis-p6-cache, 242
- Dublin Core, 234
- duplication of effort, xvi, 26

E

- embedded documentation, 154
- embedded links
 - definition of, 358
 - examples of, 47, 55
 - HTML hypermedia controls for, 111
- end-of-life plans, 188
- entity-bodies
 - definition of, 358
 - function of, 20
- entries, 93
- error messages, 295, 296
 - (see also status codes)
- ETag headers, 244, 326
- Expect headers, 326
- Expires headers, 242, 327
- extensibility, 104, 343, 353

extension link relations, 64

F

Fiat standards, xxiii, 56, 157

Fielding constraints

- definition of, 29, 341

- development of, 354

- for uniform interfaces, 345

- hypermedia and, 348

- Representational State Transfer, 348

- resource identification, 346

- resource manipulation through representations, 346

- self-descriptive messages, 347

Fielding, Roy, 29, 341

filesystems, 259

fn class, 113

FOAF, 285

<form> tags, 48, 111

From headers, 327

FTP (file transfer protocol), 14

G

GeoJSON, 225–229

GET method, 6

- as a safe method, 18, 25

- conditional GET, 242, 248

- details of, 34

- function of, 33

- in collection-based design, 100

- partial GET, 246

- pipelining, 247

- with <form> tag, 48, 111

Gopher protocol, 14

gzip, 245

H

HAL (Hypertext Application Language)

- basics of, 125, 216

HAL+JSON documents, 125

HAL+XML documents, 125

HATEOAS (hypermedia as the engine of application state)

- definition of, 358

- example of, 14

hCard format, 113

HEAD method

- details of, 40

- function of, 33

header-based content negotiation, 241

headers

- definition of, 358

- number available, 238

- (see also HTTP headers)

hMaze microformat, 114

home pages, in design process, 163, 170

Host headers, 327

HTML (HyperText Markup Language)

- adding application semantics to forms, 119

- as hypermedia format, 46, 109, 215

- benefits of, 110

- changing resource state with, 117

- class attribute, 112

- data structure in, 110

- hMaze microformat, 114

- HTML4 limitations, 124

- HTML5 advantages, 124

- hypermedia controls in, 110

- id attribute, 112

- microdata in, 116

- microformats for, 113

- plug-in application semantics for, 111

- rel attribute, 111

HTTP extensions

- LINK method, 258

- PATCH method, 257

- UNLINK method, 258

- WebDAV, 259

HTTP headers

- Accept, 239, 319

- Accept-Charset, 319

- Accept-Encoding, 245, 319

- Accept-Language, 239, 320

- Accept-Ranges, 246, 320

- Age, 320

- Allow, 320

- Authorization, 321

- Cache-Control, 242, 321

- Connection, 322

- Content-Disposition, 322

- Content-Encoding, 246, 323

- Content-Language, 323

- Content-Length, 323

- Content-Location, 218, 324

- Content-MD5, 324

- Content-Range, 324
- Content-Type, 20, 324
- Cookie, 325
- creating custom, 317
- Date, 326
- definition of, 317
- ETag, 244, 326
- Expect, 326
- Expires, 242, 327
- From, 327
- guides to, 317
- Host, 327
- If-Match, 328
- If-Modified-Since, 243, 328
- If-None-Match, 244, 329
- If-Range, 329
- If-Unmodified-Since, 329
- Last-Modified, 243, 330
- Link, 51, 330
- Link-Template, 220, 331
- Location, 218, 331
- Max-Forwards, 332
- number available, 238
- Pragma, 332
- Prefer, 333
- Preference-Applied, 333
- Proxy-Authenticate, 334
- Proxy-Authorization, 334
- Range, 247, 334
- Referer, 335
- Retry-After, 335
- Set-Cookie, 336
- Slug, 336
- TE, 336
- Trailer, 336
- Transfer-Encoding, 337
- types of, 317
- Upgrade, 338
- User-Agent, 338
- Vary, 339
- Via, 339
- Warning, 339
- WWW-Authenticate, 250, 340
- HTTP messages, overview of, 33
- HTTP methods
 - choice of, 42
 - definition of, 358
 - details on, 34–42
 - example of, 6
 - standardization of, 8
- HTTP protocols
 - avoiding data loss, 248
 - caching, 242
 - canonical URLs, 241
 - compression, 245
 - conditional requests, 242, 248
 - content negotiation and, 239
 - discouraging pointless requests, 241
 - for authentication, 249–257
 - headers, 238
 - HTTP extensions, 257
 - hypermedia menus, 240
 - look-before-you-leap (LBYL) requests, 244
 - partial GET, 246
 - pipelining, 247
 - representation choices and, 239
 - response codes, 238
 - version 1.0, 341
 - version 1.1, 238, 342, 347
 - version 2.0, 260
- HTTP requests
 - parts of, 52
 - types under HTML controls, 48
- HTTP responses
 - examples of, 53
 - number available, 238
 - parts of, 18
- HTTP sessions, 4
- HTTP verbs (see HTTP methods)
- HTTP: The Definitive Guide (Gourley and Totty), 238
- human-readable documentation, 122, 134, 149
- Hydra, 276
- hypermedia
 - definition of, 48, 237, 358
 - distributed, 343, 353
 - examples of, 14
 - Fielding constraints and, 348
 - generic hypermedia language (see HTML)
 - guiding HTTP requests, 52
 - HTML format for, 46, 109
 - HTTP responses and, 53
 - link headers and, 51, 218
 - pipelining and, 247
 - poor understanding of, xviii
 - pure-hypermedia designs, 109
 - URI Templates and, 49
 - URIs vs. URLs, 50

- usefulness of, 45, 52, 67, 185
- vs. media, 122
- vs. strings, 55
- workflow control with, 54
- hypermedia control
 - and semantic gap, 57
 - definition of, 358
 - examples of, 46
 - in World Wide Web, 354
 - uses for, 52
- hypermedia formats
 - Atom Publishing Protocol, 207
 - Collection+JSON, 206
 - collection-based, 206–215
 - Content-Location headers, 218
 - domain-specific, 200–206
 - GeoJSON, 225–229
 - HAL (Hypertext Application Language), 125, 216
 - HTML, 215
 - JSON Home Documents, 219
 - Link-Template headers, 220
 - Location headers, 218
 - Maze+XML, 200
 - OData, 208–215
 - OpenSearch, 201
 - problem detail documents, 201
 - pure hypermedia, 215–224
 - Siren, 129, 217
 - SVG (Scalable Vector Graphics), 202
 - URL lists, 219
 - variety of, 199
 - VoiceXML, 204
 - WADL (Web Application Description Language), 221
 - XForms, 223
 - XLink, 222
- hypermedia mazes
 - common steps in, 59, 66
- hypermedia menus, 240
- hypermedia-based service documents, 190

I

- id attribute, 112
- idempotent state transition
 - definition of, 358
 - examples of, 36
 - pipelining and, 247
- identification of resources, 346, 354

- If-Match headers, 328
- If-Modified-Since headers, 243, 328
- If-None-Match headers, 244, 329
- If-Range headers, 329
- If-Unmodified-Since headers, 329
- tags, 48, 111
- implementation, design process and, 170
- information resources, definition of, 30, 358
- Internet Assigned Numbers Authority (IANA), 63, 230
- Internet of Things, 287
- Internet-Drafts, xxv, 34, 40, 242, 287
- internet-scale, 344, 353
- itemdrop attribute, 116
- items
 - data bits in, 96
 - definition of, 93
 - links in, 97
 - permanent links to, 96
 - representation of, 95
- itemscope attribute, 116
- itemtype attribute, 116

J

- JSON (JavaScript Object Notation)
 - introduction to, 20
 - strings vs. links, 55
- JSON Home Documents, 219
- JSON-LD (JSON for Linking Data), 151–154, 274

L

- Last-Modified headers, 243, 330
- layered systems, in World Wide Web, 351, 354
- Link headers, 51, 218, 330
- LINK method
 - details of, 258
 - examples of, 39
 - function of, 34
- link relation
 - and ALPS, 148
 - definition of, 358
 - examples of, 5, 62
 - for profiles, 135, 139
 - in design process, 161, 171
 - in HTML, 111
 - microformats and, 115
 - Microformats wiki and, 232

- registration of, 171, 230
- registry of, 63
- types of, 64
- unsafe, 140
- <link> tags, 110
- Link-Template headers, 220, 331
- Linked Data, 272
- Location headers, 218, 331
- look-before-you-leap (LBYL) requests, 244
- low entry-barrier, 342, 353

M

- machine-readable documentation, 122, 141
- Mapmaker client, 74, 80
- Max-Forwards headers, 332
- Maze+XML example
 - API calls and, 67
 - application state in, 64
 - automated client for, 74, 76
 - collections in, 65
 - format overview, 200
 - goal of, 59
 - human-driven client, 68–72
 - link relations in, 62
 - media type of, 60
 - rat's-eye-view in, 61
 - semantic challenge and, 60
 - server for, 72
 - standard extensions, 77–80
- media types
 - definition of, 20, 358
 - in design process, 167, 170, 176, 183
 - negotiating content and, 240
 - registration of, 184
 - versioning of, 186
- members, 93
- messages, self-descriptive, 5, 347, 354
- meta data profiles, 141
- meta-vocabulary, 270
- microblogging APIs, xvii
- microdata, 116, 136
- microformats, 113, 138, 230
- MIME types (see media types)
- monitor-type API clients, 88

N

- names, reconciling in design process, 164, 174
- Node library, choice of, xxii

O

- OAuth 1.0
 - concepts behind, 252
 - drawbacks of, 255
 - features of, 254
- OAuth 2.0, 256
- odata hypermedia format, 208–215
- ontology, 270, 284
- open standards, xxiv
- OpenSearch hypermedia format, 201
- OPTIONS method
 - details of, 40
 - function of, 33
- outbound links
 - definition of, 359
 - examples of, 54
 - HTML hypermedia controls for, 110
- overloaded POST
 - definition of, 359
 - examples of, 41
- overview representations, 239

P

- pagination, 101
- PATCH method
 - details of, 38, 257
 - function of, 34
 - in collection-based design, 101
- personal credentials, 250
- personal standards, xxiv
- pipelining, 247
- POST method
 - function of, 24, 33
 - with <form> tag, 48, 111
- POST-to-append
 - definition of, 359
 - details of, 37
 - in collection-based design, 100
- Pragma headers, 332
- Prefer headers, 333
- Preference-Applied headers, 333
- problem detail documents, 201, 296
- profiles
 - ALPS (Application-Level Protocol Semantics) for, 143
 - application semantics and, 138
 - definition of, 135, 359
 - embedded documentation, 154

- in design process, 169, 176
- JSON-LD (JSON for Linking Data) for, 151
- link relations and, 139, 148
- linking to, 135
- media type parameter for, 136, 240
- protocol semantics and, 137
- publishing, 170
- semantic descriptors and, 140
- special-purpose hypermedia controls for, 136
- unsafe link relations and, 140
- versioning of, 187
- XMDP (XHTML Meta Data Profile) format, 141
- protocol semantics
 - definition of, 359
 - importance of, 33
 - in collection-based design, 100
 - profiles and, 137
 - vs. application semantics, 57
- Proxy-Authenticate headers, 334
- Proxy-Authorization headers, 334
- publication, in design process, 170
- PUT method
 - details of, 37
 - function of, 33
 - in collection-based design, 101

R

- Range headers, 247, 334
- RDF (Resource Description Framework)
 - basics of, 264
 - description vs. representation strategies and, 264, 268
 - RDF Schema, 270
 - resource types in, 269
 - URL treatment in, 265
 - usefulness of, 266
- Referer headers, 335
- registered link relations, 64
- registration, 250
- rel attribute, 111
- representation strategy, 263
- Representational State Transfer
 - architectural constraints and, 348
 - example of, 32
- representations
 - choices of, 239
 - definition of, 30, 359

- examples of, 3
- manipulation of resources through, 346, 354
- transfer of, 31
- resource design approach, xxi
- resource state, 11
 - changing with HTML, 117, 119
 - definition of, 359
 - representations of, 30
- resource types, 269
- resources
 - collection resources, 93
 - definition of, 30, 359
 - description of, 263
 - examples of, 2
 - identification of, 346, 354
 - in design process, 178
 - manipulation through representations, 346, 354
 - multiple representations of, 32, 239
 - relationships between, 54
- response codes, 4, 19, 238, 295
 - (see also HTTP responses)
 - (see also status codes)
- response headers, 20
 - (see also HTTP headers)
- RESTful architectures
 - ability to adapt to change, xvi
 - as marketing buzzword, xviii, 29
 - Fielding constraints and, 341, 348
 - HTTP headers in, 317–340
 - uniform interfaces in, 345
 - vs. SOAP-based APIs, xvii
- RESTful Web Services (Richardson and Ruby), xvii
- Retry-After headers, 335
- RFC 2616, 238
- RFCs (requests for comments), xxv, 238
- Richardson, Leonard, xvii
- Ruby, Sam, xvii

S

- safe state, definition of, 360
- schema.org, 117, 233
- schema.org RDF, 285
- <script> tags, 111
- script-type API clients, 88
- search templates, 99, 102
- self-descriptive messages, 5, 347, 354
 - definition of, 360

- semantic descriptors, 140
 - definition of, 360
 - in design process, 159, 161, 173
- semantic gap
 - dealing with, 16, 344, 354
 - definition of, 360
 - domain-specific designs and, 60, 83
 - examples of, 27
 - hypermedia controls and, 57
 - solving, 155
- semantic types, 269
- Semantic Web, 264, 286
- server implementation
 - design process and, 170
 - for Maze+XML example, 72
- service description documents, 189
- sessions, 4, 250
- Set-Cookie headers, 336
- seven-step design procedure
 - implementation, 170
 - media type choice, 167
 - overview of, 158
 - preparatory work, 158
 - profile creation, 169
 - publication, 170
 - reconciling names, 164
 - semantic descriptor list, 159
 - state diagram creation, 161
- short sessions, 4
- Siren, 129, 217
- Slug headers, 336
- snell-link-method Internet-Draft, 34, 40
- SOAP-based APIs, xvii
- software agents, 89
- standards
 - benefits/shortfalls of, xxii
 - corporate standards, xxiv
 - Fiat standards, xxiii, 56, 157
 - Internet-Drafts, xxv, 34, 40, 242, 287
 - open standards, xxiv
 - personal standards, xxiv
 - RFCs (requests for comments), xxv, 238
- state diagrams, 161, 174, 178
- state transition
 - definition of, 360
 - in HTML, 119
- statelessness
 - and authentication, 250
 - definition of, 360
 - example of, 5
 - in World Wide Web, 349, 353
 - self-descriptive message constraint and, 347
- status codes
 - 100 (Continue), 298
 - 101 (Switching Protocols), 298
 - 200 (OK), 4, 238, 295–297, 299
 - 201 (Created), 23, 299
 - 202 (Accepted), 299
 - 203 (Non-Authoritative Information), 300
 - 204 (No Content), 300
 - 205 (Reset Content), 301
 - 206 (Partial Content), 301
 - 300 (Moved Permanently), 297
 - 300 (Multiple Choices), 302
 - 301 (Moved Permanently), 302
 - 302 (Found), 302
 - 303 (See Other), 303
 - 304 (Not Modified), 243, 303
 - 305 (Use Proxy), 304
 - 306 (Unused), 304
 - 307 (Temporary Redirect), 304
 - 308 (Permanent Redirect), 305
 - 400 (Bad Request), 238, 296–297, 306
 - 400 (Not Found), 297
 - 401 (Unauthorized), 306
 - 402 (Payment Required), 307
 - 403 (Forbidden), 307
 - 404 (Method Not Allowed), 307
 - 404 (Not Found), 307
 - 406 (Not Acceptable), 308
 - 407 (Proxy Authentication Required), 308
 - 408 (Request Timeout), 308
 - 409 (Conflict), 297, 309
 - 410 (Gone), 309
 - 411 (Length Required), 310
 - 412 (Precondition Failed), 310
 - 413 (Request Entity Too Large), 310
 - 414 (Request-URL Too Long), 311
 - 415 (Unsupported Media Type), 311
 - 416 (Requested Range Not Satisfiable), 312
 - 417 (Expectation Failed), 312
 - 428 (Precondition Failed), 312
 - 429 (Too Many Requests), 312
 - 431 (Request Header Fields Too Large), 313
 - 451 (Unavailable For Legal Reasons), 313
 - 500 (Internal Server Error), 297, 314
 - 501 (Not Implemented), 314
 - 502 (Bad Gateway), 314

- 502 (Service Unavailable), 314
- 504 (Gateway Timeout), 315
- 505 (HTTP Version Not Supported), 315
- 511 (Network Authentication Required), 315
- client error (4xx), 297, 305–313
- definition of, 295
- families of, 296
- function of, 19
- informational (1xx), 296, 298–299
- problem detail documents and, 201, 296
- redirection (3xx), 297, 301–305
- server error (5xx), 297, 313–315
- successful (2xx), 297, 299–301
- usefulness of, 295

SVG (Scalable Vector Graphics), 202

T

TE headers, 336

tight coupling, 189

Trailer headers, 336

transclusion

- definition of, 360
- examples of, 55

Transfer-Encoding headers, 337

U

uniform interfaces

- definition of, 360
- in World Wide Web, 350, 354

UNLINK method

- details of, 258
- examples of, 39
- function of, 34

unsafe link relation, 140

Upgrade headers, 338

URI Templates, 49, 181

URIs (Uniform Resource Identifiers)

- and RDF documents, 265
- definition of, 360
- in World Wide Web, 354
- standards for, 342
- templates for, 49
- vs. URLs, 50
- Well-Known URI Registry, 172

URL lists, 219

URLs (Uniform Resource Locators)

- canonical URLs, 241
- definition of, 360

- in design process, 180
- in RDF documents, 265
- publishing, 170
- URL space partitions, 186
- vs. URIs, 50

User-Agent headers, 338

V

Vary headers, 339

vCard format, 84, 113

verbs (see HTTP methods)

Via headers, 339

vocab.org, 286

vocabulary, 270, 284

VoiceXML, 204

W

WADL (Web Application Description Language), 221

Warning headers, 339

web host metadata documents, 282

WebDAV, 259

WebFinger, 283

Well-Known URI Registry, 172

World Wide Web

- addressability in, 3
- application state in, 10
- architectural constraints and, 353
- architectural properties of, 342, 353
- as distributed computing, xv, 1
- caching in, 350, 353
- client-server architecture, 349
- code on demand in, 352, 354
- connectedness in, 13
- Fielding constraints and, 341
- layered systems in, 351, 354
- redirects in, 9
- resource state in, 11
- resources/representations in, 2
- self-descriptive messages in, 5
- short sessions in, 4
- standardized HTTP methods in, 8
- statelessness in, 349
- technologies underlying, 1, 29, 237
- uniform interfaces in, 350, 354
- vs. APIs, 344
- vs. competing technologies, 14

write template, 98

WSDL (Web Service Definition Language) descriptions, 189
WWW-Authenticate headers, 250, 340

X

XForms, 223

XLink, 222

XMDP (XHTML Meta Data Profile) format, 141

XRD format, 280

Y

You Type It, We Post It design example, 173

关于作者

Leonard Richardson 是 *Ruby Cookbook*(O'Reilly) 的联合作者，他还参与编写了包括 *Beautiful Soup* 在内的多个开源代码库。他是加利福尼亚人，目前定居于纽约。

Mike Amundsen 是国际知名的作家和演说家，他游历美国和欧洲进行着咨询和演讲，所涉及的主题非常宽泛，包括分布式网络架构、web 应用开发和云计算。他近期的工作则专注于超媒体在创建和维持应用这一过程中所起到的作用，而该应用是可以随着时间成功地进行进化的。在创建适应性分布式系统时，经常会引用他在 2011 年写的书——*Building Hypermedia APIs with HTML5 and Node*(O'Reilly)。在他的业余时间，Mike 会和他的家人在肯塔基州享受天伦时光。

Sam Ruby 是一位杰出的软件开发人员，他是 W3C HTML 工作组的联合主席，并且在 Apache 软件基金会的众多开源软件项目中有过显著的贡献。他还是 IBM 新兴技术组的高级技术人员。

封面介绍

RESTful Web APIs 封面上的动物是霍氏二趾树懒（学名：Choloepus hoffmanni）。霍氏二趾树懒被发现于中美洲及南美洲的雨林，并以德国自然学家 Karl Hoffmann 的名字进行命名。霍氏二趾树懒因其两个弧形的前爪而得名，它使用它们来倒挂于树枝之上。

成年二趾树懒通常体长 21~28 英寸，重约 4.6~20 磅。事实上，二趾树懒往往要花一个月的时间来消化 3 个胃里的食物，因此造成了体重的巨大变化。相较于同龄雄性二趾树懒，雌性二趾树懒的体型更大一些。雄性和雌性二趾树懒都有着棕褐色或浅棕色的皮毛，且覆盖着薄薄的绿藻。二趾树懒多数时候都吃叶子，同时也吃果实和花朵。

二趾树懒是一种夜间活动的生物，多数时间都栖息于树上。与很多树懒一样，二趾树懒是出了名的行动缓慢，但它的缓慢是为了配合低能量的食性。与其他同属树懒科的物种一样，它们的视觉和听觉很差，很容易成为肉食动物的目标。

封面图片取自 *Cassell's Natural History*。封面字体是 Adobe ITC Garamond。正文字体是 Adobe Minion Pro；标题字体是 Adobe Myriad Condensed；代码字体是 Dalton Maag's Ubuntu Mono。